

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



- 由浅入深地介绍Keras与TensorFlow深度学习类神经网络
- 使用实际的数据集配合范例程序代码介绍各种深度学习算法，并示范如何进行数据预处理、训练数据、建立模型和预测结果

TensorFlow + Keras

林大贵 著

深度学习人工智能实践应用

人工智能时代必须学习的新技术



清华大学出版社





TensorFlow+ Keras

深度学习人工智能实践应用

林大贵 著

清华大学出版社
北 京

内 容 简 介

本书提供安装、上机操作指南,同时辅以大量范例程序介绍 TensorFlow + Keras 深度学习方面的知识。本书分 9 部分,共 21 章,内容主要包括基本概念介绍、TensorFlow 与 Keras 的安装、Keras MNIST 手写数字识别、Keras CIFAR-10 照片图像物体识别、Keras 多层感知器预测泰坦尼克号上旅客的生存概率、使用 Keras MLP、RNN、LSTM 进行 IMDb 自然语言处理与情感分析、以 TensorFlow 张量运算仿真神经网络的运行、TensorFlow MNIST 手写数字识别、使用 GPU 大幅加快深度学习训练。

TensorFlow + Keras 深度学习方面的知识不需要具备高等数学模型、算法等专业知识,读者只需要具备基本的 Python 程序设计能力,按照本书的步骤循序渐进地学习,就可以了解深度学习的基本概念,进而实际运用深度学习的技术。

本书为博硕文化股份有限公司授权出版发行的中文简体字版本

北京市版权局著作权合同登记号:图字 01-2017-6576

本书封面贴有清华大学出版社防伪标签,无标签者不得销售

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

TensorFlow+Keras 深度学习人工智能实践应用 / 林大贵著. —北京:清华大学出版社,2018(2018.4重印)
ISBN 978-7-302-49302-0

I. ①T… II. ①林… III. ①人工智能—算法—研究 IV. ①TP18

中国版本图书馆 CIP 数据核字(2018)第 004552 号

责任编辑:夏毓彦

封面设计:王翔

责任校对:闫秀华

责任印制:杨艳

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

印装者:三河市金元印装有限公司

经 销:全国新华书店

开 本:190mm×260mm

印 张:20.75

字 数:531 千字

版 次:2018 年 2 月第 1 版

印 次:2018 年 4 月第 2 次印刷

印 数:3001~5000

定 价:69.00 元

产品编号:077469-01

序

深度学习是人工智能/机器学习研究中的一个新领域，它的概念源于人工神经网络的研究，目的在于建立含有多个隐含层的多层感知器的一种深度学习结构，以此来模拟人脑进行分析学习的神经网络。深度学习在视觉识别（人脸识别或面部表情识别）、图像识别、语音识别、文字识别和生物医学等方面的应用中具有优势，而且已经取得了非常好的效果。

对于具有人工智能和机器学习基本概念的读者或者开发者，想“更上一层楼”掌握深度学习理论和技术，并希望可以在实践中快速运用深度学习技术，那么本书就非常合适。对于开设了“深度学习”人工智能课程的高等院校，本书可以作为学生上机实践的实验用书。

本书介绍的 TensorFlow+Keras 深度学习与类神经网络不需要读者具有高等数学建模的专业知识，也不需要具有计算机专业深厚的算法知识和精专的程序设计技能，读者只需要具有基本的 Python 程序设计能力，而后按照本书的章节循序渐进地学习，就可以了解深度学习的基本概念，而且可以按照书中的详细步骤来运用本书提供的大量范例程序，通过上机实践操作来实际实现深度学习技术。

本书的范例程序不需要修改就可以直接运行，而且同时支持 Windows 环境和 Linux Ubuntu 环境，读者只要按照书中的说明下载和安装好 CUDA、cuDNN、TensorFlow GPU 版本与 Keras 即可。

本书附录 A 提供了下载全部范例程序的网址和步骤，这些范例程序使用实际的数据集来实现各种深度学习的算法，并示范如何进行数据预处理、训练数据、建立模型、预测结果，展示如何把 Keras 与 TensorFlow 深度学习类神经网络运用到具体的应用中。

深度学习会用到大数据技术，而且本书的范例要用到 Python 程序设计语言，本书并

未在这些方面深入讲解，建议需要学习这方面内容的读者去寻找相关出版物，以便能更好地结合本书来学习和实践深度学习技术，从而完善自己在人工智能知识方面的结构。

资深架构师 赵军

2017 年 11 月

前言

近年来，人工智能（Artificial Intelligence，AI）吸引了大众与媒体的目光，AlphaGo 的成功更加让人工智能技术变得炙手可热，其实 AI 早已进入了我们的生活，如手机中的语音助理、人脸识别、影音平台每日的推荐等。然而，人工智能的发展才刚刚起步，未来人工智能的应用将会深入生活的每一个层面，也就是说未来一定是 AI 的时代。

深度学习是人工智能中成长最快的领域，深度学习就是仿真人类神经网络的工作方式，常见的深度学习架构有深度神经网络（DNN）、卷积神经网络（CNN）、递归神经网络（RNN）等。深度学习特别适用于视觉识别、语音识别、自然语言处理、识别癌细胞等领域，目前已经取得非常好的效果。

近年来，各大科技公司（如 Google、Microsoft、Facebook、Amazon、Tesla 等）都积极投入到深度学习领域进行研发。以 Google 公司为例，它在 2014 年以 4 亿美元并购了 DeepMind 公司。2016 年，由 DeepMind 开发的人工智能围棋程序 AlphaGo 以 4:1 击败了世界级围棋冠军李世石，引起了网络与媒体的注目，让人们了解到深度学习的威力强大。

TensorFlow 最初由 Google Brain Team 团队开发，Google 使用 TensorFlow 进行研究及开发自身产品，并于 2015 年公开了它的源代码，所有的开发者都可以免费使用。Google 的很多产品早就使用了机器学习或深度学习，例如 Gmail 过滤垃圾邮件、Google 语音识别、图像识别、翻译等。

TensorFlow 功能强大，执行效率高，支持各种平台。但是，TensorFlow 是比较底层的深度学习链接库，学习门槛高，对于从未接触过深度学习的初学者，如果一开始就学习 TensorFlow，就要面对其特殊的“计算图”（computational graph）程序设计模式，还必须自行设计张量（Tensor）运算，可能会有很大的挫折感。

所以本书先介绍 Keras，这是以 TensorFlow 为底层的、高级的深度学习链接库，可以很容易地建立深度学习模型，进行训练并使用模型预测，对初学者而言学习门槛较低。等



读者熟悉了深度学习模型后，再去学习 TensorFlow 就很轻松了。

近年来，深度学习和人工智能技术快速发展的一个很重要的因素是，GPU 提供了强大的并行运算架构，可以让深度学习训练比用 CPU 来进行要快数十倍。本书也特别介绍了 GPU 的安装与应用，读者只需要有 NVIDIA 显示适配器（显卡），然后按照本书的介绍依次安装 CUDA、cuDNN、TensorFlow GPU 版本与 Keras，就可以使用 GPU 大幅加快深度学习的训练。

林大贵

本书章节与范例程序介绍

本书的目标读者

使用 TensorFlow+Keras 深度学习与类神经网络不需要具备高等数学模型、算法等专业知识，读者只需要具有基本的 Python 程序设计能力，按照本书的步骤循序渐进地学习，就可以了解深度学习的基本概念，进而实际运用深度学习的技术。

本书的特色是提供安装、上机操作指南，同时辅以大量范例程序。

➤ 上机操作

本书详细介绍了如何在 Windows 与 Linux Ubuntu 系统上安装 Anaconda、TensorFlow 与 Keras，用于深度学习的程序设计，并且详细介绍了安装 CUDA、cuDNN、TensorFlow GPU 版本与 Keras 的步骤，用于加快深度学习的训练速度。

➤ 范例程序

以实际范例程序来学习程序设计是最有效率的方式。因此本书使用实际的数据集配合范例程序代码来介绍各种深度学习算法，并示范如何进行数据预处理、训练数据、建立模型、预测结果，由浅入深地介绍 Keras 与 TensorFlow 深度学习类神经网络。

本书章节内容及上机实践操作与范例程序介绍

➤ 基本概念介绍

章节	章节名称	说明
1	人工智能、机器学习与深度学习简介	介绍人工智能、机器学习、深度学习等基本原理与概念
2	深度学习的原理	介绍如何以矩阵数学公式来仿真类神经网络的运行
3	TensorFlow 与 Keras 介绍	介绍 TensorFlow 与 Keras 的基本原理、概念与应用

➤ TensorFlow 与 Keras 的安装

章节	章节名称	说明
4	在 Windows 中安装 TensorFlow 与 Keras	上机操作 介绍如何在 Windows 中建立 Anaconda 虚拟环境，并且安装 TensorFlow 与 Keras
5	在 Linux Ubuntu 中安装 TensorFlow 与 Keras	上机操作 介绍如何在 Ubuntu Linux 操作系统中安装 TensorFlow 与 Keras

➤ Keras MNIST 手写数字识别

章节	章节名称	说明
6	Keras MNIST 手写数字识别数据集	范例程序 Keras_Mnist_Introduce.ipynb 介绍如何下载与读取 MNIST 手写数字数据集，并且进行数据预处理
7	Keras 多层感知器识别手写数字	范例程序 Keras_Mnist_MLP_h256.ipynb 使用多层感知器 MLP 模型识别 MNIST 数据集的手写数字 范例程序 Keras_Mnist_MLP_h1_1000.ipynb 将 MLP 模型加宽，以提高准确率 范例程序 Keras_Mnist_MLP_h1_1000_DropOut.ipynb 将 MLP 模型加宽并加入 DropOut 以避免过度拟合，提高准确率 范例程序 Keras_Mnist_MLP_h1000_DropOut_h1000_DropOut.ipynb 将 MLP 模型加宽、加深，以提高准确率
8	Keras 卷积神经网络识别手写数字	范例程序 Keras_Mnist_CNN.ipynb 使用卷积神经网络识别 MNIST 数据集中的手写数字，其分类精度接近 0.99

➤ Keras CIFAR-10 照片图像物体识别

章节	章节名称	说明
9	Keras CIFAR-10 图像识别数据集	范例程序 Keras_Cifar_CNN_Introduce.ipynb 介绍下载、读取与数据预处理 CIFAR-10 图像识别数据集，共 10 种分类：飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船、卡车
10	Keras 卷积神经网络识别 CIFAR-10 图像	范例程序 Keras_Cifar_CNN.ipynb 使用卷积神经网络识别 CIFAR-10 图像 范例程序 Keras_Cifar_CNN_DeepConv3.ipynb 将 CNN 模型加宽、加深，以 3 次的卷积与池化运算提高准确率

➤ Keras 多层感知器预测泰坦尼克号上旅客的生存概率

章节	章节名称	说明
11	Keras 泰坦尼克号上的旅客数据集	范例程序 Keras-Taianic_Introduce.ipynb 介绍如何使用 Python Pandas 下载、读取泰坦尼克号的数据集，并且介绍泰坦尼克号数据集的特色，进行数据的预处理
12	Keras 多层感知器预测泰坦尼克号上旅客的生存概率	范例程序 Keras-Taianic_MLP.ipynb 建立多层感知器模型，预测泰坦尼克号上旅客的生存概率，并预测《泰坦尼克号》电影男女主角的生存概率，找出泰坦尼克号上其他旅客的感人故事

➤ 使用 Keras MLP、RNN、LSTM 进行 IMDb 自然语言处理与情感分析

章节	章节名称	说明
13	IMDb 网络电影数据集与自然语言处理	范例程序 Keras_Imdb_Introduce.ipynb 介绍如何以 Keras 下载、读取 IMDb 网络电影，并进行自然语言处理



章节	章节名称	说明
14	Keras 建立 MLP、RNN、LSTM 模型进行 IMDb 情感分析	范例程序 Keras_Imdb_MLP.ipynb 使用 Keras 建立多层感知器模型进行 IMDb 情感分析 范例程序 Keras_Imdb_MLP_Large.ipynb 在自然语言处理时，加入更多字数的字典与影评文字长度，以提高准确率 范例程序 Keras_Imdb_RNN.ipynb 使用 Keras 建立递归神经网络模型进行 IMDb 情感分析 范例程序 Keras_Imdb_LSTM.ipynb 使用 Keras 建立长短期记忆模型进行 IMDb 情感分析

➤ 以 TensorFlow 张量运算仿真神经网络的运行

章节	章节名称	说明
15	TensorFlow 程序设计模式	范例程序 TensorFlow_Basic.ipynb 介绍 TensorFlow 程序设计模式的概念以及张量的基本运算 范例程序 TensorFlow_Board_area.ipynb 示范如何以可视化的方式来查看所建立的“计算图”
16	以 TensorFlow 张量运算仿真神经网络的运行	范例程序 TensorFlow_Tensor_neural.ipynb 介绍以 TensorFlow 张量（矩阵）运算来模拟类神经网络的运行，并且建立 layer 函数，后续可以用于构建多层感知器

➤ TensorFlow MNIST 手写数字识别

章节	章节名称	说明
17	TensorFlow MNIST 手写数字识别数据集	范例程序 TensorFlow_Mnist_Introduce.ipynb 介绍如何以 TensorFlow 下载、读取 MNIST 手写数字数据集
18	TensorFlow 多层感知器识别手写数字	范例程序 TensorFlow_Mnist_MLP_h256.ipynb 使用 TensorFlow 构建多层感知器模型，识别 MNIST 数据集的手写数字 范例程序 TensorFlow_Mnist_MLP_h1000.ipynb 将 MLP 模型加宽，以提高准确率 范例程序 TensorFlow_Mnist_MLP_h1000-h1000.ipynb 将 MLP 模型加深、加宽，以提高准确率
19	TensorFlow 卷积神经网络识别手写数字	范例程序 TensorFlow_Mnist_CNN.ipynb 示范如何使用 TensorFlow 卷积神经网络完成手写数字识别功能

➤ 使用 GPU 大幅加快深度学习训练

章节	章节名称	说明
20	TensorFlow GPU 版本的安装	上机操作 示范如何建立 Anaconda 虚拟环境并安装 Nvidia CUDA、cuDNN、TensorFlow GPU 版本、Keras

章节	章节名称	说明
21	使用 GPU 加快 TensorFlow 与 Keras 训练	<p>范例程序 Test_GPU.ipynb 示范如何在 TensorFlow GPU 执行环境中实际测试 CPU 与 GPU 执行效率的差异</p> <p>范例程序 TensorFlow_Mnist_MLP_h1000.ipynb 使用这个实际范例测试 TensorFlow 程序在 CPU 与 GPU 中执行效率的差异</p> <p>范例程序 Keras_Cifar_CNN.ipynb 使用这个实际范例测试 Keras 程序在 CPU 与 GPU 中执行效率的差异</p>

➤ 本书范例程序下载

下载网址: <https://pan.baidu.com/s/1c2rXnH2> (注意区分数字和英文字母大小写)。

如果下载有问题, 请发送电子邮件至 booksaga@126.com, 邮件主题设置为“求 TensorFlow+Keras 深度学习人工智能实践应用范例程序”。

➤ 指令整理

安装 Tensor 与 Keras 必须使用 Windows 或 Linux 指令。为了方便读者练习, 将这些指令整理在与本书有关的微博中。读者可以从微博中复制这些指令, 然后粘贴到 Windows “命令提示符”窗口或 Linux “终端”程序中, 以节省打字的时间, 避免输错字母或字符。微博网址如下:

<https://www.weibo.com/hadoopsparkbook>

目 录

第 1 章	人工智能、机器学习与深度学习简介	1
1.1	人工智能、机器学习、深度学习的关系	2
1.2	机器学习介绍	4
1.3	机器学习分类	4
1.4	深度学习简介	7
1.5	结论	8
第 2 章	深度学习的原理	9
2.1	神经传导的原理	10
2.2	以矩阵运算仿真神经网络	13
2.3	多层感知器模型	14
2.4	使用反向传播算法进行训练	16
2.5	结论	21
第 3 章	TensorFlow 与 Keras 介绍	22
3.1	TensorFlow 架构图	23
3.2	TensorFlow 简介	24
3.3	TensorFlow 程序设计模式	26
3.4	Keras 介绍	27
3.5	Keras 程序设计模式	28
3.6	Keras 与 TensorFlow 比较	29
3.7	结论	30
第 4 章	在 Windows 中安装 TensorFlow 与 Keras	31
4.1	安装 Anaconda	32



4.2	启动命令提示符	35
4.3	建立 TensorFlow 的 Anaconda 虚拟环境	37
4.4	在 Anaconda 虚拟环境安装 TensorFlow 与 Keras	40
4.5	启动 Jupyter Notebook	42
4.6	结论	48
第 5 章	在 Linux Ubuntu 中安装 TensorFlow 与 Keras	49
5.1	安装 Anaconda	50
5.2	安装 TensorFlow 与 Keras	52
5.3	启动 Jupyter Notebook	53
5.4	结论	54
第 6 章	Keras MNIST 手写数字识别数据集	55
6.1	下载 MNIST 数据	56
6.2	查看训练数据	58
6.3	查看多项训练数据 images 与 label	60
6.4	多层感知器模型数据预处理	62
6.5	features 数据预处理	62
6.6	label 数据预处理	64
6.7	结论	65
第 7 章	Keras 多层感知器识别手写数字	66
7.1	Keras 多元感知器识别 MNIST 手写数字图像的介绍	67
7.2	进行数据预处理	69
7.3	建立模型	69
7.4	进行训练	73
7.5	以测试数据评估模型准确率	77
7.6	进行预测	78
7.7	显示混淆矩阵	79
7.8	隐藏层增加为 1000 个神经元	81
7.9	多层感知器加入 DropOut 功能以避免过度拟合	84
7.10	建立多层感知器模型包含两个隐藏层	86
7.11	结论	89

第 8 章 Keras 卷积神经网络识别手写数字	90
8.1 卷积神经网络简介	91
8.2 进行数据预处理	97
8.3 建立模型	98
8.4 进行训练	101
8.5 评估模型准确率	104
8.6 进行预测	104
8.7 显示混淆矩阵	105
8.8 结论	107
第 9 章 Keras CIFAR-10 图像识别数据集	108
9.1 下载 CIFAR-10 数据	109
9.2 查看训练数据	111
9.3 查看多项 images 与 label	112
9.4 将 images 进行预处理	113
9.5 对 label 进行数据预处理	114
9.6 结论	115
第 10 章 Keras 卷积神经网络识别 CIFAR-10 图像	116
10.1 卷积神经网络简介	117
10.2 数据预处理	118
10.3 建立模型	119
10.4 进行训练	123
10.5 评估模型准确率	126
10.6 进行预测	126
10.7 查看预测概率	127
10.8 显示混淆矩阵	129
10.9 建立 3 次的卷积运算神经网络	132
10.10 模型的保存与加载	135
10.11 结论	136
第 11 章 Keras 泰坦尼克号上的旅客数据集	137
11.1 下载泰坦尼克号旅客数据集	138
11.2 使用 Pandas DataFrame 读取数据并进行预处理	140



11.3	使用 Pandas DataFrame 进行数据预处理	142
11.4	将 DataFrame 转换为 Array	143
11.5	将 ndarray 特征字段进行标准化	145
11.6	将数据分为训练数据与测试数据	145
11.7	结论	147
第 12 章	Keras 多层感知器预测泰坦尼克号上旅客的生存概率	148
12.1	数据预处理	149
12.2	建立模型	150
12.3	开始训练	152
12.4	评估模型准确率	155
12.5	加入《泰坦尼克号》电影中 Jack 与 Rose 的数据	156
12.6	进行预测	157
12.7	找出泰坦尼克号背后的感人故事	158
12.8	结论	160
第 13 章	IMDb 网络电影数据集与自然语言处理	161
13.1	Keras 自然语言处理介绍	163
13.2	下载 IMDb 数据集	167
13.3	读取 IMDb 数据	169
13.4	查看 IMDb 数据	172
13.5	建立 token	173
13.6	使用 token 将“影评文字”转换成“数字列表”	174
13.7	让转换后的数字长度相同	174
13.8	结论	176
第 14 章	Keras 建立 MLP、RNN、LSTM 模型进行 IMDb 情感分析	177
14.1	建立多层感知器模型进行 IMDb 情感分析	178
14.2	数据预处理	179
14.3	加入嵌入层	180
14.4	建立多层感知器模型	181
14.5	训练模型	182
14.6	评估模型准确率	184
14.7	进行预测	185
14.8	查看测试数据预测结果	185

14.9	查看《美女与野兽》的影评	187
14.10	预测《美女与野兽》的影评是正面或负面的	190
14.11	文字处理时使用较大的字典提取更多文字	192
14.12	RNN 模型介绍	193
14.13	使用 Keras RNN 模型进行 IMDb 情感分析	195
14.14	LSTM 模型介绍	197
14.15	使用 Keras LSTM 模型进行 IMDb 情感分析	199
14.16	结论	200
第 15 章	TensorFlow 程序设计模式	201
15.1	建立“计算图”	202
15.2	执行“计算图”	204
15.3	TensorFlow placeholder	206
15.4	TensorFlow 数值运算方法介绍	207
15.5	TensorBoard	208
15.6	建立一维与二维张量	211
15.7	矩阵基本运算	212
15.8	结论	214
第 16 章	以 TensorFlow 张量运算仿真神经网络的运行	215
16.1	以矩阵运算仿真神经网络	216
16.2	以 placeholder 传入 X 值	220
16.3	创建 layer 函数以矩阵运算仿真神经网络	222
16.4	建立 layer_debug 函数显示权重与偏差	225
16.5	结论	226
第 17 章	TensorFlow MNIST 手写数字识别数据集	227
17.1	下载 MNIST 数据	228
17.2	查看训练数据	229
17.3	查看多项训练数据 images 与 labels	232
17.4	批次读取 MNIST 数据	234
17.5	结论	235
第 18 章	TensorFlow 多层感知器识别手写数字	236
18.1	TensorFlow 建立多层感知器辨识手写数字的介绍	237

18.2	数据准备	239
18.3	建立模型	239
18.4	定义训练方式	242
18.5	定义评估模型准确率的方式	243
18.6	进行训练	244
18.7	评估模型准确率	249
18.8	进行预测	249
18.9	隐藏层加入更多神经元	250
18.10	建立包含两个隐藏层的多层感知器模型	251
18.11	结论	252
第 19 章 TensorFlow 卷积神经网络识别手写数字		253
19.1	卷积神经网络简介	254
19.2	进行数据预处理	255
19.3	建立共享函数	256
19.4	建立模型	258
19.5	定义训练方式	264
19.6	定义评估模型准确率的方式	264
19.7	进行训练	265
19.8	评估模型准确率	266
19.9	进行预测	267
19.10	TensorBoard	268
19.11	结论	270
第 20 章 TensorFlow GPU 版本的安装		271
20.1	确认显卡是否支持 CUDA	273
20.2	安装 CUDA	274
20.3	安装 cuDNN	278
20.4	将 cudnn64_5.dll 存放的位置加入 Path 环境变量	281
20.5	在 Anaconda 建立 TensorFlow GPU 虚拟环境	283
20.6	安装 TensorFlow GPU 版本	285
20.7	安装 Keras	286
20.8	结论	286

第 21 章 使用 GPU 加快 TensorFlow 与 Keras 训练	287
21.1 启动 TensorFlow GPU 环境	288
21.2 测试 GPU 与 CPU 执行性能	293
21.3 超出显卡内存的限制	296
21.4 以多层感知器的实际范例比较 CPU 与 GPU 的执行速度	297
21.5 以 CNN 的实际范例比较 CPU 与 GPU 的执行速度	299
21.6 以 Keras Cifar CNN 的实际范例比较 CPU 与 GPU 的执行速度	302
21.7 结论	304
附录 A 本书范例程序的下载与安装说明	305
A.1 在 Windows 系统中下载与安装范例程序	306
A.2 在 Ubuntu Linux 系统中下载与安装范例程序	310

第1章

人工智能、机器学习 与深度学习简介

近年来，人工智能（Artificial Intelligence, AI）吸引了大众与媒体的目光，AlphaGo 的成功让人工智能技术变得更加炙手可热，其实 AI 早已进入你的生活，例如我们手机中的语音助理、人脸识别、影音平台的每日推荐等。然而，人工智能的发展才刚刚起步，未来人工智能的应用将会深入生活中的每一个层面，也就是说未来一定是 AI 的时代。

1.1 人工智能、机器学习、深度学习的关系

人工智能、机器学习、深度学习的关系整理如图 1-1 所示。

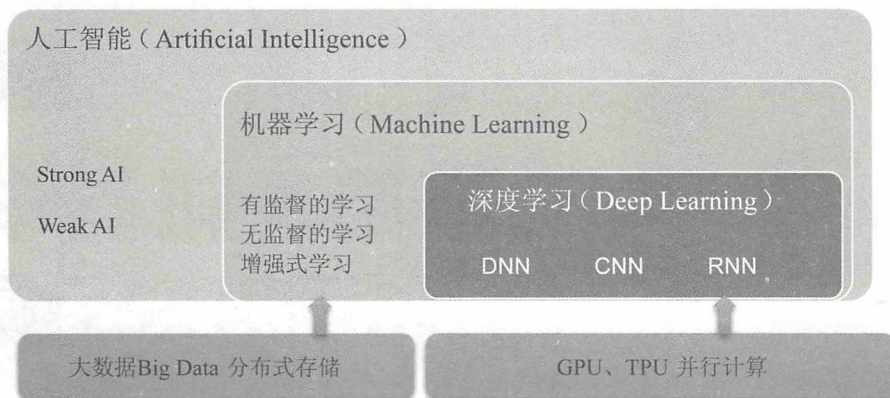


图 1-1

➤ 人工智能

“人工智能”一词最早是在 20 世纪 50 年代提出的。人工智能的目标是让计算机像人一样思考与学习。被视为人工智能之父的图灵（Alan Mathison Turing）提出了著名的“图灵测试”（Turing Testing）：人类与机器通过电传设备对话，如果人类无法根据这个对话过程判断对方是机器或人，通过图灵测试就可以认定这台机器具有人工智能。

20 世纪 80 年代，约翰·塞尔（John Searle）提出了对“人工智能”的分类方式。

- 强人工智能（Strong AI）：机器具有与人类一样完整的认知能力。
- 弱人工智能（Weak AI）：机器不需要具有与人类一样完整的认知能力，只要设计得看起来像具有智慧即可。

➤ 机器学习

机器学习（Machine Learning）是人工智能的分支。机器学习是通过算法，使用大量数据进行训练，训练完成后会产生模型。将来有新的数据时，我们可以使用训练产生的模型进行预测。机器学习可分为有监督的学习（Supervised Learning）、无监督的学习（Unsupervised Learning）和增强式学习（Reinforcement Learning）。

机器应用相当广泛，例如推荐引擎、定向广告、需求预测、垃圾邮件过滤、医学诊断、自然语言处理、搜索引擎、诈骗侦测、证券分析、视觉识别、语音识别、手写识别，等等。



➤ 深度学习

深度学习 (Deep Learning) 也是机器学习的分支。深度学习是人工智能中增长最快的领域,深度学习仿真人类神经网络的工作方式,常见的深度学习架构有多层感知器 (Multi-Layer Perceptron)、深度神经网络 (Deep Neural Network, DNN)、卷积神经网络 (Convolutional Neural Network, CNN)、递归神经网络 (Recurrent Neural Network, RNN)。深度学习已经应用于视觉识别、语音识别、自然语言处理、生物医学等领域,并且取得了非常好的效果。

➤ 为何近年来人工智能发展加速

早在 20 世纪 60 年代及 70 年代,科学家就提出了各种机器学习的算法,然而受限于当时计算机的计算能力以及大量数据的获取也不容易,因而机器学习一直都没有很成功。

● 大数据 (Big Data) 分布式存储与运算

随着全球设备、机器和系统的相互连接,从而产生了大量数据,再加上分布式存储 (例如 Hadoop、NoSQL 等) 的发展,也提供了大量数据,而且大量服务器的并行计算功能 (例如 Spark 等) 提供了巨大的运算能力。大量数据与运算能力就像燃料,推动了机器学习与深度学习的加速发展。

● GPU、TPU 并行计算

GPU (Graphics Processing Unit, 图形处理器) 原本用来处理画面像素的运算,例如电脑游戏界面所需要的大量图形运算。

CPU 与 GPU 架构上有着根本的不同: CPU 含有数颗核心,为顺序处理进行优化,而 GPU 可以有高达数千个小型而且高效率的核心,可以发挥并行计算的强大功能,如图 1-2 所示。

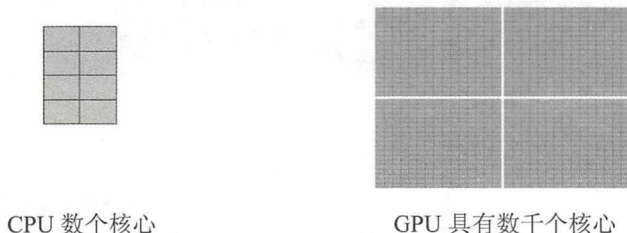


图 1-2

深度学习以大量矩阵运算模拟神经元的工作方式,矩阵运算的特性是,单一运算都很简单,但是需要大量运算,特别适合采用并行计算。GPU 通过大量核心进行并行计算,可通过 GPU 进行深度学习训练比通过 CPU 进行深度学习训练快 10~75 倍,让训练的时间从数周缩短为数天。

而 Google 公司更在 2016 年宣布,研发人工智能专用芯片 TPU (Tensor Processing Unit, 张量处理单元或张量处理芯片) 来进行并行计算。TPU 是专为深度学习特定用途设计的特殊规格的逻辑芯片 (IC),使得深度学习的训练速度更快。

1.2 机器学习介绍

机器学习的训练数据是由 features、label 组成的。

- **features:** 数据的特征，例如湿度、风向、风速、季节、气压。
- **label:** 数据的标签，也就是我们希望预测的目标，例如降雨（0：不会下雨、1：会下雨）、天气（1：晴天、2：雨天、3：阴天、4：下雪）、气温。

如图 1-3 所示，机器学习可分为两个阶段：

- 训练（Training）

训练数据是过去累积的历史数据，可能是文本文件、数据库或其他来源的数据，经过 Feature Extraction（特征提取）产生 features（数据特征）与 label（真实的标签数据），然后经过机器学习算法训练后产生模型。

- 预测（Predict）

新输入数据，经过 Feature Extraction（特征提取）产生 features（数据特征），使用训练完成的模型进行预测，最后产生预测结果。

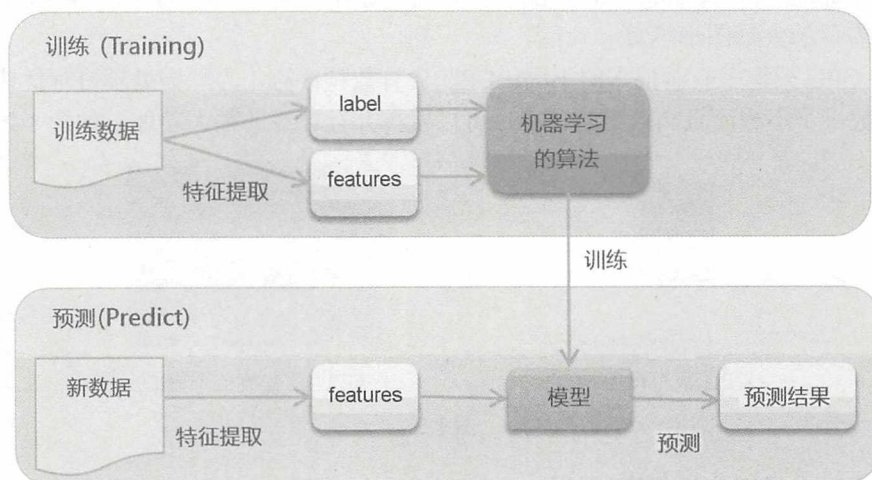


图 1-3

1.3 机器学习分类

前面介绍过，机器学习可分为有监督的学习、无监督的学习和增强式学习。以下详细介



绍其分类。

➤ 有监督的学习

有监督的学习的数据具备特征（features）与预测目标（label），通过算法训练并建立模型。当有新的数据时，我们就可以使用模型进行预测。有下列分类：

● 二元分类

已知湿度、风向、风速、季节、气压等数据特征，希望预测当天是否会下雨（0：不会下雨、1：会下雨）。希望预测的目标 label 有两个选项，就好像在做是非题。

● 多元分类

已知湿度、风向、风速、季节、气压等数据特征，希望预测当天天气（1：晴天、2：雨天、3：阴天、4：下雪）。希望预测的目标 label 有多个选项，就好像在做选择题。

● 回归分析

已知湿度、风向、风速、季节、气压等数据特征，希望预测当天气温。希望预测的目标 label 是连续的值，就好像在做计算题。

➤ 无监督的学习

对于无监督的学习，从现有数据我们不知道要预测的答案，所以没有 label（预测目标）。例如，cluster 集群算法将数据分成几个差异性最大的群组，而群组内的则相似程度最高。

➤ 增强式学习

增强式学习的原理：借助定义动作（Actions）、状态（States）、奖励（Rewards）的方式不断训练机器循序渐进，学会执行某项任务的算法。例如，训练机器玩《超级玛丽》电子游戏，动作：左/右/跳，状态：当前游戏的界面，奖励：得分/受伤，借助不断地训练学会玩游戏。常见的算法有 Q-Learning、TD（Temporal Difference）、Sarsa。

➤ 机器学习分类整理

机器学习分类整理见表 1-1。

表 1-1 机器学习分类整理

分类	细分类	特征	预测目标
有监督的学习	二元分类 (Binary Classification)	湿度、风向、风速、 季节、气压……	只有 0 与 1 选项（是非题） 0：不会下雨、1：会下雨

(续表)

分类	细分类	特征	预测目标
有监督的学习	多元分类 (Multi-Class Classification)	湿度、风向、风速、 季节、气压……	有多个选项(选择题) 1: 晴天、2: 雨天、3: 阴天、
有监督的学习	回归分析(Regression)	湿度、风向、风速、 季节、气压……	值是数值(计算题) 温度范围可能是-50~50 度
无监督的学习	群集(Clustering)	湿度、风向、风速、 季节、气压……	无 Label Cluster 集群分析, 目的是将 数据按照特征分成几个差异性最大的 群组, 而群组内的则相似程度最 高
增强式学习	Q-Learning、TD (Temporal Difference)		增强式学习的原理: 借助定义动 作、状态、奖励的方式不断训练机 器循序渐进, 学会执行某项任务的 算法, 常用于动态系统及机器人控 制等

➤ 机器学习分类

机器学习分类如图 1-4 所示。

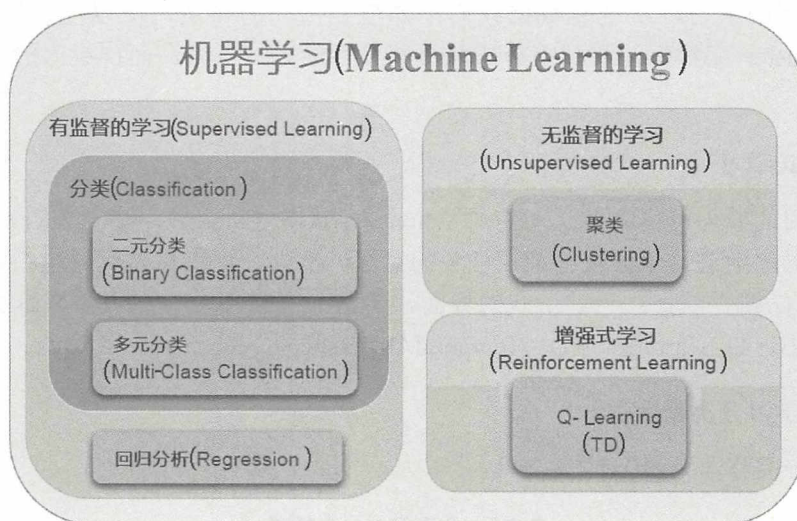


图 1-4

本书主要介绍深度学习, 如果你对机器学习与大数据有兴趣, 可以参考笔者的另一本书《Python+Spark 2.0+Hadoop 机器学习与大数据实战》。

1.4 深度学习简介

人脑的重量大约为一千多克，结构非常复杂，预估具有 860 亿个神经元以及超过 100 兆条神经相连，形成的网络比最先进的超级计算机还要强大。

因为人类神经网络太过复杂，为了方便用计算机来仿真神经网络，人们将神经元分为多个层次。通常会有一个输入层、一个输出层，隐藏层可以有非常多层（见图 1-5），所以称为深度学习。

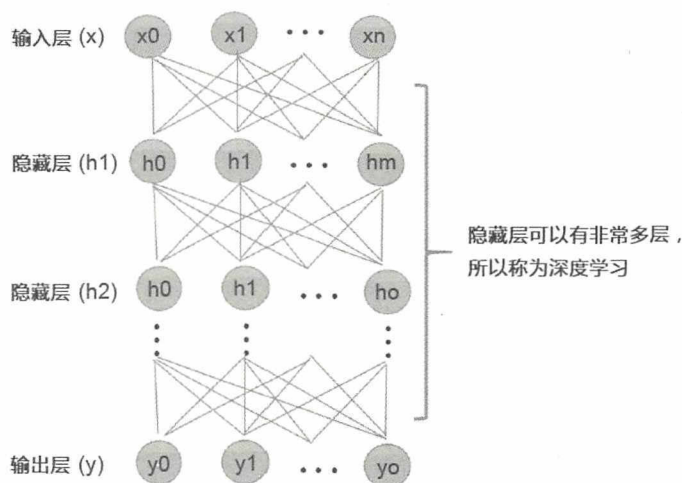


图 1-5

➤ 机器学习与深度学习的关系

如图 1-6 所示，深度学习的应用很广泛，你可以将深度学习技术应用在有监督的学习、无监督的学习和增强式学习等领域。

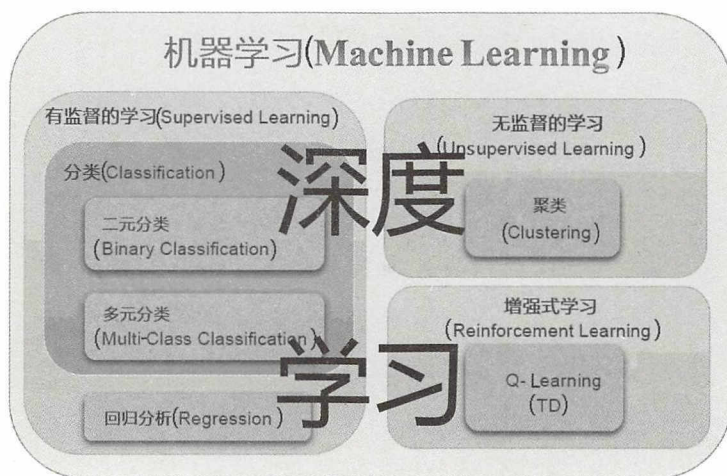


图 1-6

1.5 结论

本章我们介绍了人工智能、机器学习、深度学习的关系以及它们的基本概念，并且简明介绍了机器学习的分类，下一章将详细介绍深度学习的原理。

第2章

深度学习的原理

本章将介绍深度学习与类神经网络的原理，并介绍如何以矩阵的数学公式来仿真类神经网络的运行。

2.1 神经传导的原理

1. 神经元的信息传导

神经传导的工作原理很复杂，在此我们只是简略介绍其概念，如图 2-1 所示。

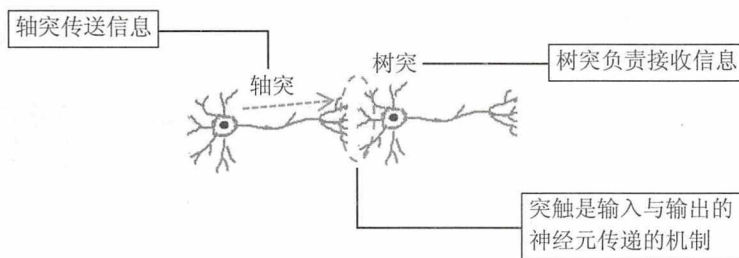


图 2-1

- **轴突传送信息**：神经元长出一个细长条的轴突，以电流方式将信息传递给另一个神经元。轴突最长可达一米，最短只有几十分之一毫米。
- **树突接收信息**：树突主要的功能是接收其他神经元传来的电化学信息，再传递给本身的细胞。
- **突触是输入与输出的神经元传递的机制**：输入与输出的神经元之间发展出的特殊结构称为突触，神经元通过释放化学物质来传递信息，当电压达到临界值时，就会通过轴突传递电脉冲动作电位至接收神经元。

2. 以数学公式仿真神经元的信息传导

为了将神经元的传导用计算机来仿真，我们必须将神经元的传导以数学公式来表示。以图 2-2 来说明如何以数学公式模拟神经元的信息传导。

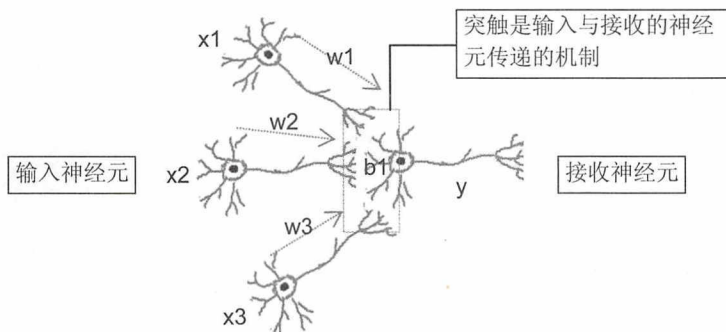


图 2-2

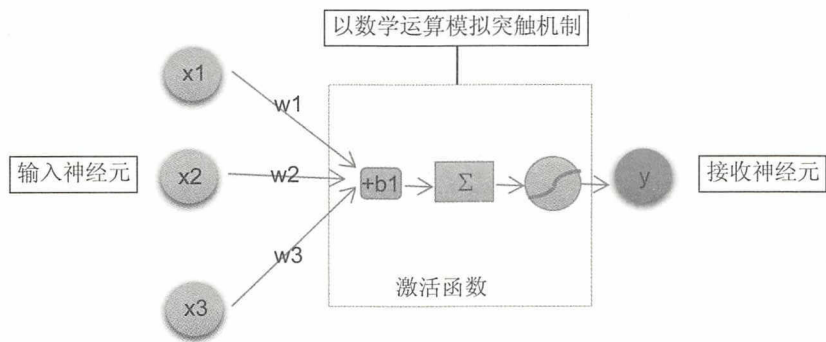


图 2-2（续）

图 2-2 可以整理为以下公式：

$$y = \text{activation function}(x1 \times w1 + x2 \times w2 + x3 \times w3 + b1)$$

详细说明如表 2-1 所示。

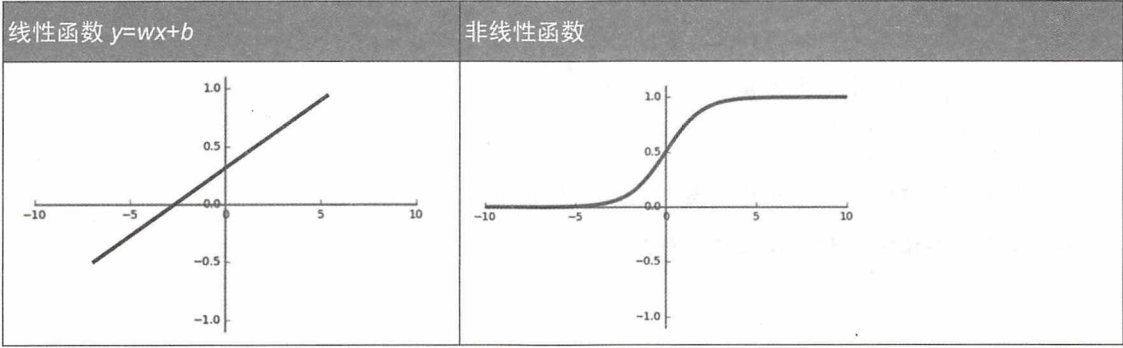
表 2-1 以数学公式模拟神经元的信息传导参数说明

项目	说明
输入 x	x 仿真输入神经元，接收外界传送信息，共有 3 个输入神经元： $x1$ 、 $x2$ 、 $x3$
接收 y	模拟接收神经元，只有一个接收神经元： y
权重 w (weight)	权重 w 模拟轴突，连接输入与接收神经元，负责传送信息，共有 3 个轴突： $w1$ 、 $w2$ 、 $w3$
偏差值 b (bias)	偏差值 b 仿真突触的结构，代表接收神经元容易被活化的程度，偏差值越高，越容易被活化并传递信息。因为接收神经元只有一个，所以也只有一个偏差值： $b1$
激活函数 (activation function)	激活函数仿真神经传导的工作方式，当接收神经元接收刺激的总和： $x1 \times w1 + x2 \times w2 + x3 \times w3 + b$ 经过激活函数的运算大于临界值时，会传递至下一个神经元。常见的激活函数如 Sigmoid 或 ReLU

3. 激活函数通常为非线性函数

以上激活函数可以仿真神经传导的工作方式将上一层神经元信号传递到下一层。激活函数通常为非线性函数，加入激活函数能让神经网络处理比较复杂的非线性问题。表 2-2 所示为线性函数与非线性函数的图像。

表 2-2 线性函数与非线性函数的图像



Keras 与 TensorFlow 支持很多激活函数，不过在此我们介绍最常用的两种：Sigmoid 与 ReLU。

4. Sigmoid 激活函数

常用的激活函数 Sigmoid 公式如下：

$$f(x) = \frac{1}{1+e^{-x}}$$

Sigmoid 激活函数的图像如图 2-3 所示。

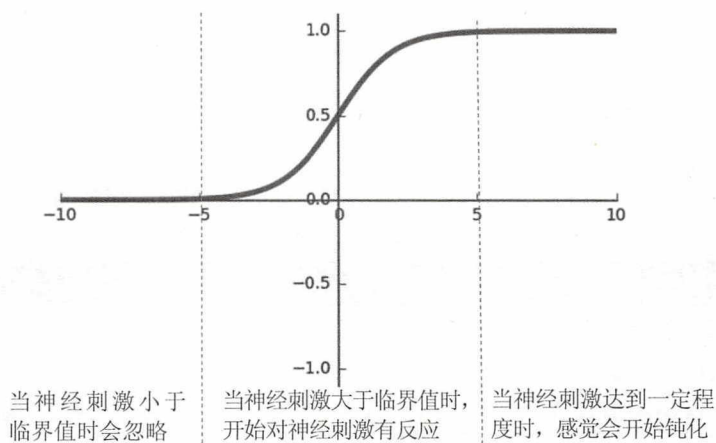


图 2-3

Sigmoid 激活函数其实与人类感觉神经对信号的接收类似，例如，当接收神经元所接收刺激的总和：

- 小于临界值时，会忽略此刺激 当 $x < -5$ 时，输出 y 接近 0。
- 大于临界值时，开始接收神经刺激 当 x 范围在 -5 与 5 之间时，随着 x 数值加大， y 的数值也加大。
- 达到一定程度时，感觉会开始钝化 即使受到更大的刺激，感觉仍维持不变，即当 $x > 5$ 时， y 的数值趋近于 1。

5. ReLU 激活函数

另一个很常见的激活函数 ReLU 的图像如图 2-4 所示。

当接收神经元所接收刺激的总和：

- 小于临界值时，会忽略此刺激 输入 x 小于 0， y 值是 0。
- 大于临界值时，开始接收神经刺激 输入 x 大于 0， y 等于 x 。

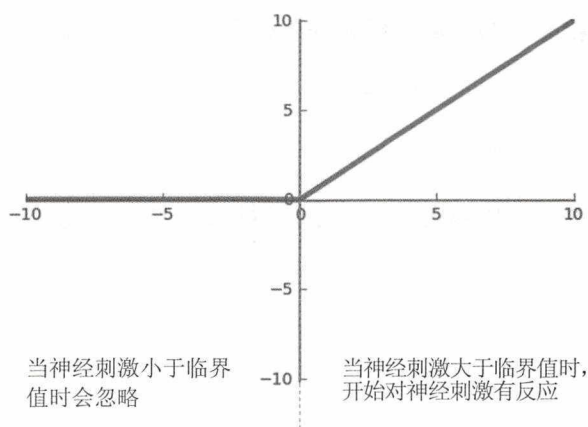


图 2-4

2.2 以矩阵运算仿真神经网络

前面只是以数学运算模拟单个接收神经元，本节将介绍以矩阵模拟两个接收神经元。

1. 以矩阵运算仿真神经网络的信息传导

多个接收神经元的神经网络如图 2-5 所示。

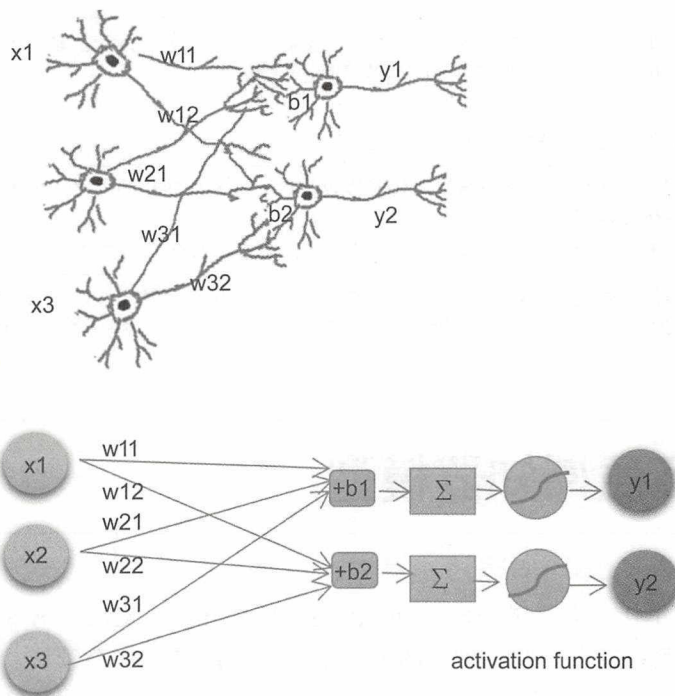


图 2-5

➤ 以数学公式模拟输出与接收神经元的工作方式：

$$y1 = \text{activation function}(x1 \times w11 + x2 \times w21 + x3 \times w31 + b1)$$

$$y2 = \text{activation function}(x1 \times w12 + x2 \times w22 + x3 \times w32 + b2)$$

➤ 以上两个数学公式可以整合成一个矩阵运算公式：

$$[y1 \ y2] = \text{activation} \left([x1 \ x2 \ x3] \times \begin{bmatrix} w11 & w12 \\ w21 & w22 \\ w31 & w32 \end{bmatrix} \right) + [b1 \ b2]$$

➤ 另一种形式的矩阵公式表示如下：

$$y = \text{activation}(x \times w + b)$$

➤ 矩阵公式以中文表示如下：

输出=激活函数(输入×权重+偏差)

说明见表 2-3。

表 2-3 以数学公式模拟输出与接收神经元的工作方式参数说明

项目	说明
输入 x	x 仿真输入神经元，接收外界传送的信息，共有 3 个输入神经元： $x1$ 、 $x2$ 、 $x3$
接收 y	模拟接收神经元，共有两个输出神经元： $y1$ 、 $y2$
权重 w	权重模拟神经元的轴突，连接输入与接收神经元，负责传送信息。因为要完全连接输入与接收神经元，共需要(输入 3)×(接收 2)=6 个轴突 $w11$ 、 $w21$ 、 $w31$ 负责传送信息给 $y1$ $w12$ 、 $w22$ 、 $w32$ 负责传送信息给 $y2$
偏差值 b	偏差值 b 仿真突触的结构，代表接收神经元容易被活化的程度，偏差值越高，越容易被活化并传递信息 如图 2-5 所示，因为接收神经元有两个，所以也有两个偏差值： $b1$ 、 $b2$
激活函数	激活函数仿真神经传导的工作方式。例如：当接收神经元 $y1$ ，接收刺激的总和($x1 \times w11 + x2 \times w21 + x3 \times w31 + b1$)经过激活函数的运算大于临界值时，会传递到下一个神经元

2.3 多层感知器模型

在 20 世纪 80 年代，多层感知器（Multilayer Perceptron, MLP）模型是一种受欢迎的机器学习解决方案，尤其是在语音识别、图像识别和机器翻译等多个领域。到 20 世纪 90 年代，MLP 遇到来自更简单的模型（例如，支持向量机（Support Vector Machine, SVM））的强烈竞争。近年来，由于深度学习的成功，多层感知器又重新受到业界的重视。

1. 以多层感知器模型识别 MNIST 手写数字图像

我们将以多层感知器模型识别 MNIST 手写数字图像说明多层感知器模型的工作方式，如图 2-6 所示。

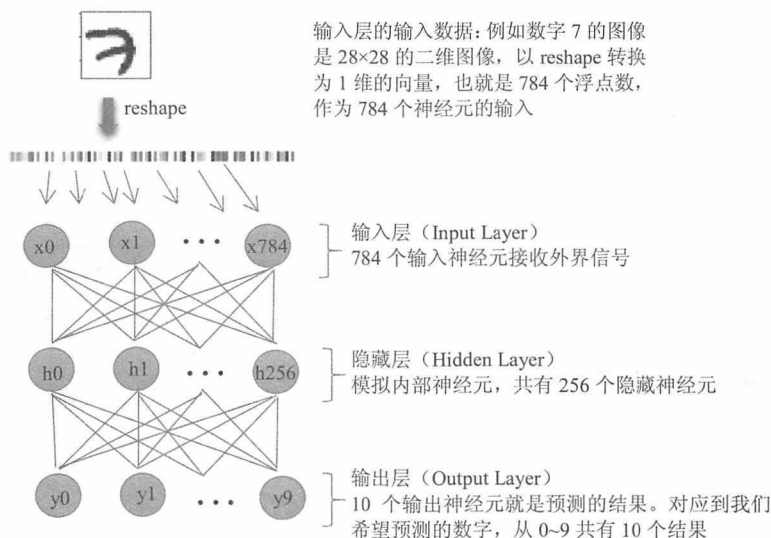


图 2-6

2. 以矩阵公式仿真多层感知器模型的工作方式 (见图 2-7)

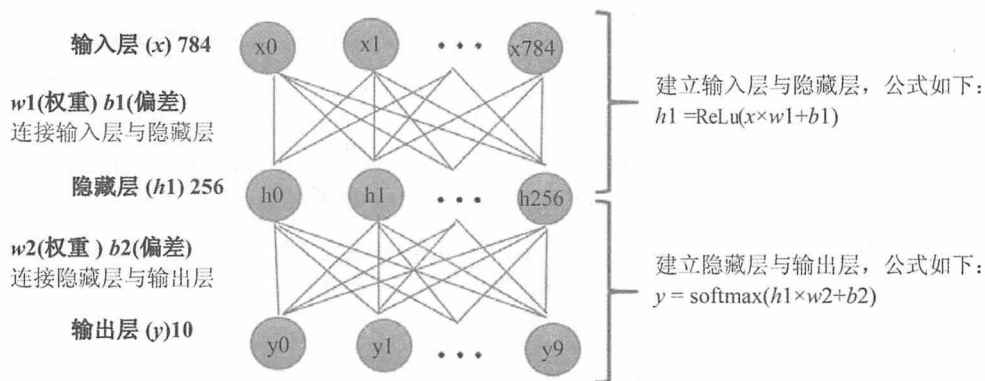


图 2-7

➤ 建立输入层与隐藏层的公式：

$$h1 = \text{ReLU}(x \times w1 + b1)$$

详细说明如表 2-4 所示。

表 2-4 输入层与隐藏层的公式参数说明

项目	说明
输入层 x	x 仿真输入神经元接收外界传送信息，如图 2-7 所示，共有 784 个神经元

(续表)

项目	说明
隐藏层 $h1$	隐藏层 $h1$ 模拟内部神经元，共有 256 个隐藏神经元
权重 $w1$	权重模拟神经元的轴突，连接输入与接收神经元，负责传送信息。连接输入层（784 个神经元）与隐藏层（256 个神经元），为了让两层的每一个神经元都互相连接，总共需要 $784 \times 256 = 200704$ 个轴突。所以 $w1$ （权重）必须是 784×256 的矩阵，用来模拟这些轴突的功能
偏差值 $b2$	偏差值 $b2$ 仿真突触的结构，代表接收神经元容易被活化的程度，偏差值越高，越容易被活化并传递信息。如图 2-7 所示，因为隐藏层共有 256 个神经元，所以偏差值是长度为 256 的向量
激活函数	激活函数仿真神经传导的工作方式，在此我们使用 ReLU 激活函数接收刺激的总和： $(x \times w1 + b1)$ ，经过激活函数 ReLU 的运算，大于临界值时，会传递至下一个神经元

➤ 建立隐藏层与输出层公式：

$$y = \text{softmax}(h1 \times w2 + b2)$$

详细说明如表 2-5 所示。

表 2-5 隐藏层与输出层的公式参数说明

项目	说明
隐藏层 $h1$	隐藏层 $h1$ 模拟内部神经元，共有 256 个隐藏神经元
输出层 y	模拟输出神经元，就是预测的结果，共有 10 个输出神经元。对应到我们希望预测的数字，从 0 到 9 共有 10 个结果
权重 $w2$	权重模拟神经元的轴突，连接输入与接收神经元，负责传送信息。连接隐藏层（256 个神经元）与输出层（10 个神经元），为了让两层的每一个神经元都互相连接，总共需要 $256 \times 10 = 2560$ 个轴突。所以 $w2$ （权重）必须是 256×10 的矩阵，用来模拟这些轴突的功能
偏差值 b	偏差值 b 仿真突触的结构，代表接收神经元容易被活化的程度，偏差值越高，越容易被活化并传递信息。如图 2-7 所示，因为接收神经元是输出层（10 个神经元），所以偏差值是长度为 10 的向量
激活函数	在输出层中，我们使用 softmax 激活函数，接收刺激的总和 $(w2 \times h1 + b2)$ 经过 softmax 运算后的输出是一个概率分布，共有 10 个输出，数值越高代表概率越高，例如输出结果由 0 算起第 7 个数字数值最高，代表预测结果是 7

2.4 使用反向传播算法进行训练

当我们建立深度学习模型后，就必须进行训练。反向传播法（Back Propagation）是训练

人工神经网络的常见方法，并且与优化器（Optimizer，例如梯度下降法）结合使用。反向传播是一种有监督的学习方法，必须输入 features（特征值）与 label（真实的值）。

简单来说，反向传播算法就是“从错误中学习”。多层感知器模型识别 MNIST 手写数字图像，过程整理如图 2-8 所示。

1. 训练前必须先进行“数据预处理”与“建立模型”

（1）数据预处理：MNIST 数据集经过数据预处理产生 features（数字图像的特征值）、label（数字图像真实的值），用于后续训练使用。

（2）建立模型：建立多层感知模型，并且以随机数初始化模型的权重（Weight）与偏差（Bias）： $w1$ 、 $b1$ ， $w2$ 、 $b2$ 。

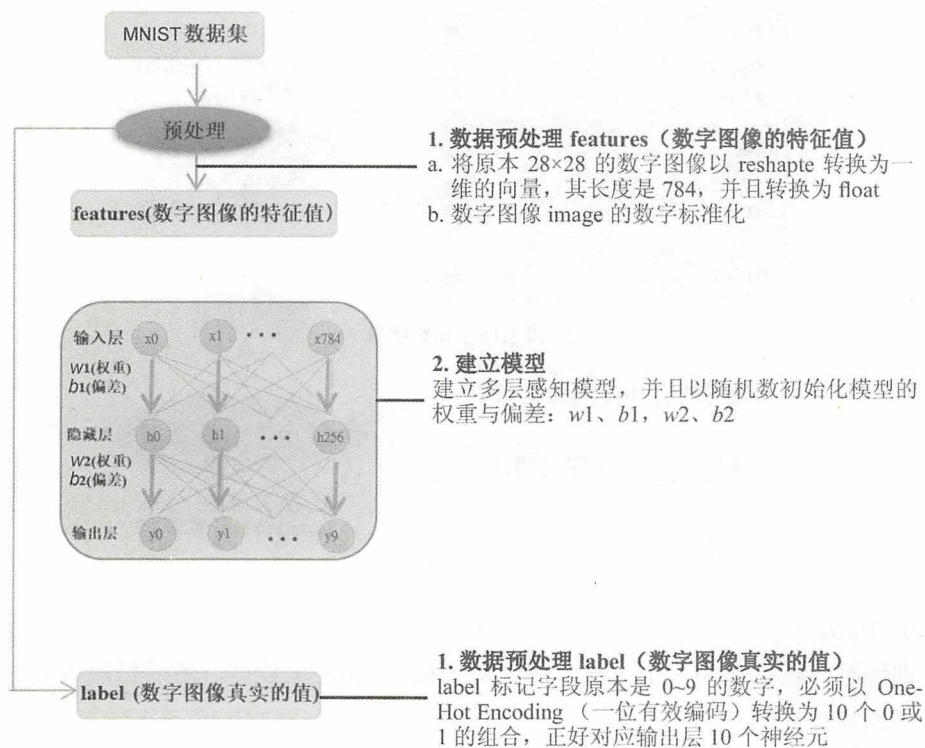


图 2-8

2. 反向传播算法训练多层感知器模型（见图 2-9）

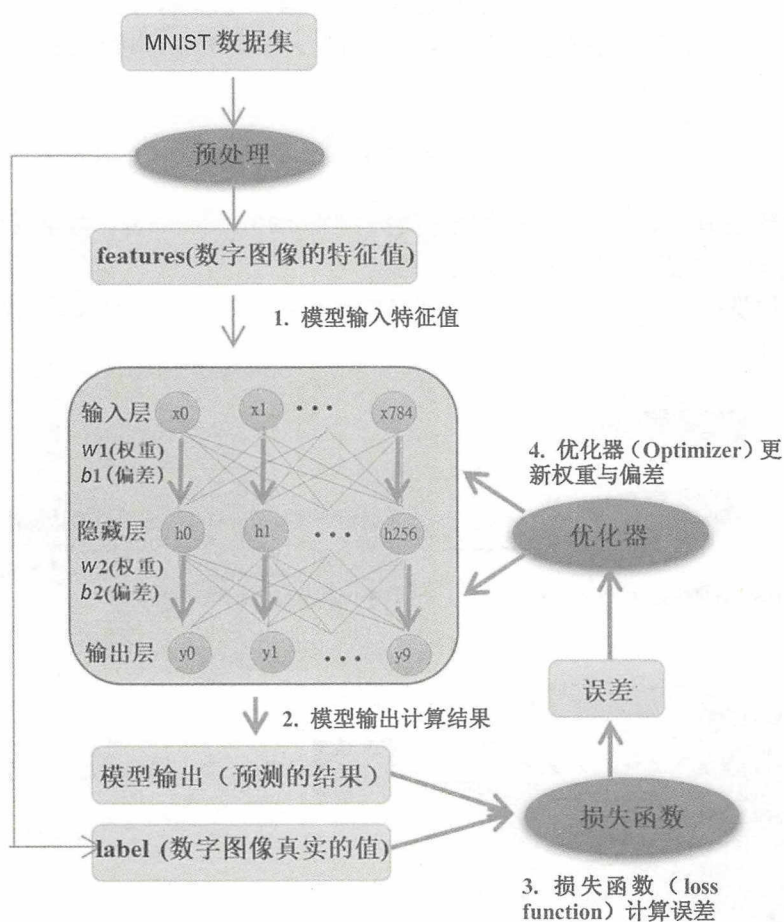


图 2-9

图 2-9 说明如下：

进行训练时，数据分为多个批次，例如每一批次 200 项数据，然后每次读取一批次数据进行反向传播算法训练。

重复以下传播和权重更新（weight update），直到误差（loss）收敛。

➤ 传播

(1) **模型输入特征值**：features（数字图像的特征值），输入到神经网络进行计算。

(2) **模型输出计算结果**：经过多个隐藏层网络进行计算，逐层向前传播，最后到达输出层，产生神经网络的输出。

➤ 权重更新

(1) **损失函数计算误差**：使用损失函数计算模型输出（预测的结果）与 label（数字图像真实的值）之间的误差值。

(2) **优化器更新权重与偏差**: 按照误差值更新神经元连接的权重与偏差, 尽量使损失函数的误差值最小化。

3. 损失函数

简单地说, 反向传播算法就是“从错误中学习”, 而损失函数就是帮我们计算误差。Cross Entropy 是深度学习常用的损失函数, 说明如图 2-10 所示。

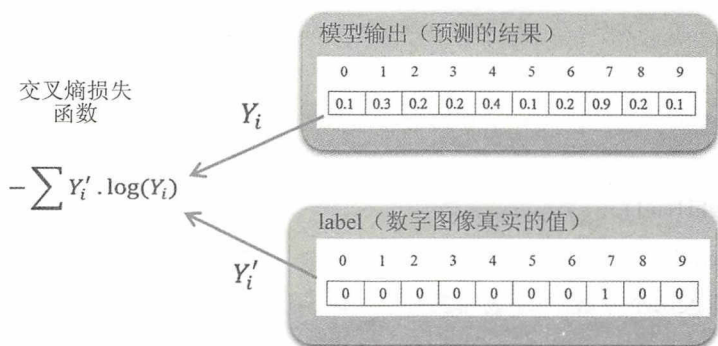


图 2-10

以上损失函数计算模型输出 (预测的结果) 与 label (数字图像真实的值) 之间的误差值, 都是以 One-Hot Encoding (一位有效编码) 表示, 例如预测数字 7。

- **label (数字图像真实的值)**: 由 0 算起第 7 个数字是 1, 其他都是 0。
- **模型输出 (预测的结果)**: 预测结果 0 的概率是 0.1 (10%), 预测结果 1 的概率是 0.3 (30%), 等等。预测结果 7 的概率是 0.9, 代表预测结果数字 7 有 90% 的概率, 其他的概率都不高。

4. 优化器

优化器就是使用某种数值方法在不断的批次训练中不断更新权重与偏差, 使损失函数的误差值最小化, 并最终找出误差值最小的“权重与偏差的组合”。

在深度学习中, 通常使用随机梯度下降法 (Stochastic Gradient Descent, SGD) 来对“权重与偏差的组合”进行优化。随机梯度下降法可以想象成在所有“权重与偏差的组合”组成的高维度空间中, 每个训练批次沿着每个维度下降的方向走一小步, 经过许多次步骤, 就可以找到优化的“权重与偏差的组合”。

➤ 二维权重与损失函数

真实的深度学习中权重与偏差数量很多, 会形成非常多维的空间。

为了方便说明, 我们假设最简单的情况, 即只有两个权重 weight1 (w_1) 与 weight2 (w_2)。因为只有两个权重, 所以可以把 w_1 与 w_2 画成如图 2-11 所示的二维图形。

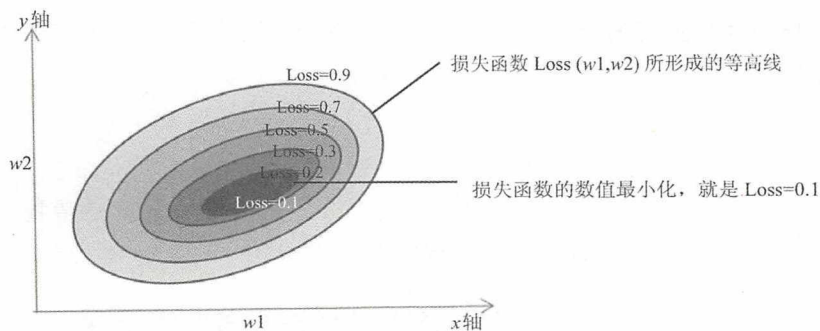


图 2-11

图 2-11 说明如下:

- x 轴是 $\text{weight1}(w1)$ 。
- y 轴是 $\text{weight2}(w2)$ 。
- 损失函数输入两个参数: $w1$ 与 $w2$ 。

按照 $\text{Loss}(w1, w2)$ 的数值可以画出由多个椭圆形所组成的等高线, 颜色越深, 代表 Loss 数值越小。

➤ SGD 梯度下降法

SGD 梯度下降法就是每次沿着 Loss 下降的方向每个训练批次走一小步, 经过许多次训练步骤就能够下降到 $\text{Loss}=0.1$, 损失函数的误差最小化, 如图 2-12 所示。

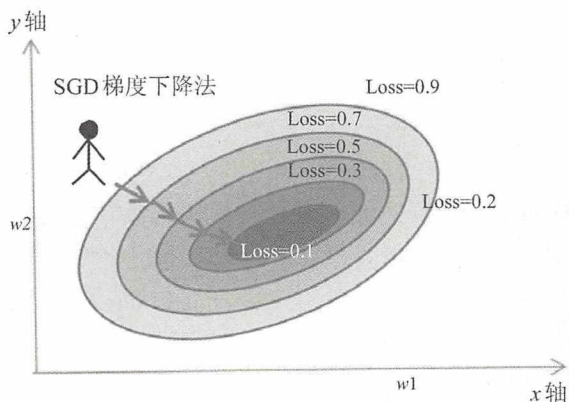


图 2-12

梯度下降法, 用比较通俗的说法就是: 一个在山上的人正在寻找山谷最低点 (试图找到损失函数 Loss 的极小值)。因为大雾能见度低, 看不见下山的道路, 所以他必须利用局部信息来找到极小值。他使用梯度下降法, 观察当前位置处的陡峭程度 (梯度), 然后沿着 (下降梯度) 最大的方向前进。使用此方法不断前进, 最终找到山谷最低点。

另外, 还有许多随机梯度下降法的变形, 如 RMSprop、Adagrad、Adadelata、Adam 等, 适用于不同的深度学习模型。



不同的优化器具有不同的训练效果，你可以参考这个网页的说明：

<http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

此网页上的图以动态方式显示出不同的优化器，以不同的路径找到 Loss 的最小值。

2.5 结论

本章我们介绍了深度学习类神经网络原理，并介绍了如何以矩阵数学公式来仿真类神经网络的运行。接下来将介绍 TensorFlow 与 Keras，可以让我们构建各种深度的学习模型。如果读者有些地方还看不懂，没有关系，只需要有基本概念，后续章节配合程序看就会清楚了。

第3章

TensorFlow与Keras介绍

本章将介绍 TensorFlow 与 Keras 的概念和程序设计模式。TensorFlow 功能强大，执行效率高，支持各种平台，然而属于低级的深度学习链接库，学习门槛高。所以本书先介绍 Keras，它是高级的深度学习链接库，对初学者学习门槛低，可以很容易地建立深度学习模型，并且进行训练、预测。等读者熟悉深度学习模型概念之后，再来学习 TensorFlow 就比较轻松了。

在第2章中，我们学习了深度学习的核心概念，是以张量（矩阵）运算模拟神经网络的。因此 TensorFlow 的主要设计就是让矩阵运算达到最高性能，并且能够在各种不同的平台执行。TensorFlow 最初由 Google（谷歌）公司的 Brain Team 团队开发，Google 公司使用 TensorFlow 进行研究及自身产品开发，并于 2015 年 11 月公开了源代码，在 Apache 2.0 与开放源代码规范下，所有的开发者都可以免费使用。Google 的很多产品早就使用了机器学习或深度学习，例如 Gmail 过滤垃圾邮件、Google 语音识别、Google 图像识别、Google 翻译等。

TensorFlow 功能强大又好用，Google 为什么要开放源代码呢？Google 认为机器学习是创新技术，而且是未来技术的关键，这方面的研究是全球性而且增长快速的，但是缺乏共同的标准。Google 希望通过开源社区的分享建立一个庞大的社区，并且建立共同的标准，这样可以扩展各种深度学习的应用领域，让 TensorFlow 更加完善。

3.1 TensorFlow 架构图

TensorFlow 架构图说明如图 3-1 所示。

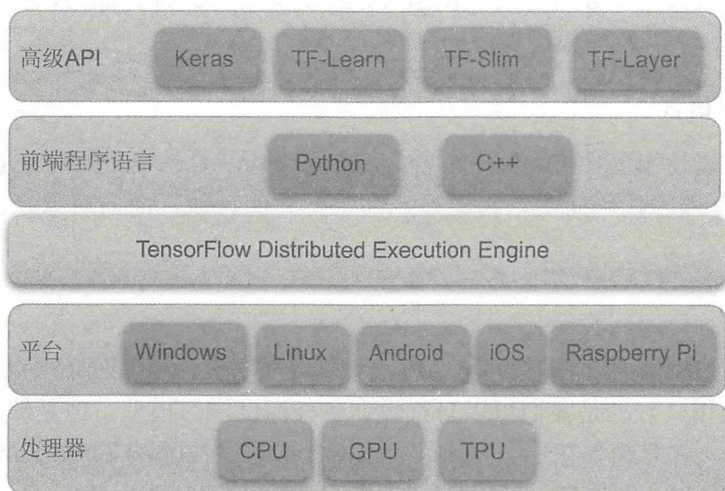


图 3-1

参照以上的架构图，我们自下而上进行说明。

➤ 处理器：TensorFlow 可以在 CPU、GPU、TPU 中执行

- **CPU**：每一台计算机都有中央处理器（CPU），可以执行 TensorFlow。
- **GPU**：图形处理器，GPU 含有高达数千个微小且高效率的计算内核，可以发挥并行计算的强大功能。



- **TPU**: TPU (Tensor Processing Unit) 是 Google 为人工智能 (AI) 研发的专用芯片, 比 GPU 的计算性能更佳, 不过目前只部署在 Google 数据中心, 也许未来会对外销售。

➤ 平台: TensorFlow 具备跨平台能力, 可以在目前主流的平台执行

TensorFlow 可以在不需修改程序代码的前提下, 让深度学习模型在不同的平台上执行训练, 以提升效率。

- **Windows**: 个人计算机最常用的操作系统, 让初学者也可以使用。
- **Linux**: TensorFlow 可以在各种版本的 Linux 操作系统中执行。
- **Android**: 在 Android 上运行 TensorFlow 可以让深度学习进入移动端, Android 设备已达到十几亿台, 设备的执行性能也日益提升, 越来越适合运行 TensorFlow。
- **iOS**: TensorFlow 可以在 iOS 移动设备或 Mac OS 中执行。
- **Raspberry Pi**: 树莓派可以用于开发物联网或机器人, 提供人工智能功能。
- **云端执行**: 可以通过云端大量的服务器加速深度学习模型的训练。

➤ TensorFlow Distributed Execution Engine (分布式执行引擎)

在深度学习中, 最花时间的就是模型的训练, 尤其大型的深度学习模型必须使用大量数据进行训练, 需要数天乃至数周之久, TensorFlow 具备分布式计算能力, 可同时在数百台机器上执行训练模型, 大幅缩短模型训练的时间。

➤ 前端程序语言

TensorFlow 可以使用多种前端程序语言, 例如 Python、C++等, 但对 Python 的支持是最好的, Python 具有程序代码简明、易学习、高生产力的特质, 面向对象、函数式的动态语言, 应用非常广泛。

➤ 高级 API

TensorFlow 是比较低级的深度学习 API, 所以用程序设计模型时必须自行设计: 张量乘积、卷积等底层操作, 好处是我们可以自行设计各种深度学习模型, 但是缺点是开发时需要编写更多程序代码, 并且需要花更多时间。所以网上的开发社区以 TensorFlow 为底层开发很多高级的深度学习 API, 例如 Keras、TF-Learn、TF-Slim、TF-Layer 等。

这样让开发者使用更简洁、更可读性的程序代码就可以构建出各种复杂的深度学习模型。本书主要介绍 Keras, 因为 Keras 的功能最完整。

3.2 TensorFlow 简介

TensorFlow 是由 Tensor 与 Flow 所组成的, 说明如下:



➤ Tensor (张量)

在数学里,张量是一种几何实体或广义上的“数量”,在此“数量”包含“标量、向量或矩阵”。零维的张量为标量,一维的张量是向量,二维以上的张量是矩阵,如图 3-2 所示。

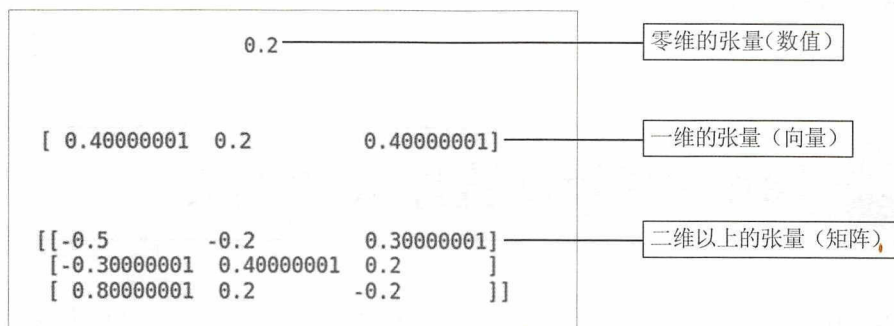


图 3-2

➤ Flow (数据流)

想象当你到了一个陌生国度,你不会当地的语言,为了到达目的地,最好的方式是画一张地图,告诉司机你要去的目的地,司机则会按照你提供的地图载你前往。

TensorFlow 也是相同的概念,为了让 TensorFlow 可以支持不同的程序设计语言接口,并且让 TensorFlow 程序可以在各种平台执行,所有的 TensorFlow 程序都是先建立“计算图”(computational graph),这是张量运算和数据处理的流程。

我们可以使用 TensorFlow 提供的模块以不同的程序设计语言建立“计算图”。TensorFlow 提供的模块非常强大,我们可以设计张量运算流程,并且构建各种深度学习或机器学习模型。建立“计算图”完成后,我们就可以在不同的平台上执行“计算图”。

如图 3-3 所示,这是典型的“计算图”,功能很简单,其算式为 $y = \text{MatMul}(x, w) + b$ (x 、 w 、 b 都是张量, w 与 b 先使用随机数进行初始化,使用 MatMul 将 x 与 w 进行张量乘积,再加上 b ,最后结果是 y)。

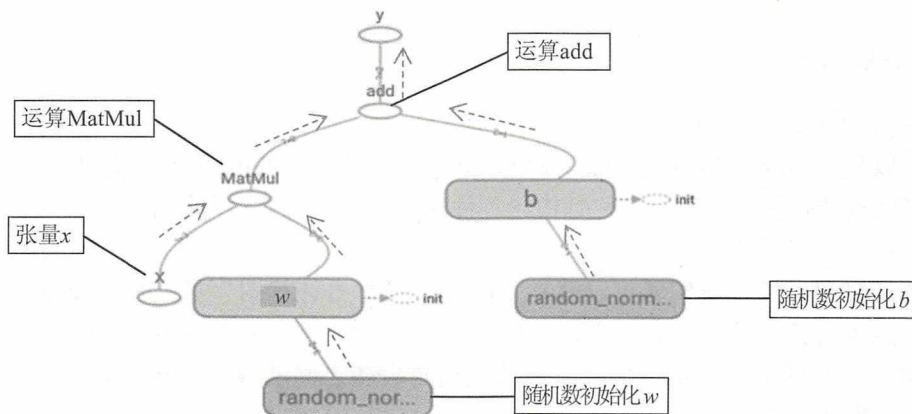


图 3-3

在上面的“计算图”中：节点（Node）代表运算，而边（Edge）代表张量的数据流。我们可以想象边就是管线，张量（数据）在管线中流动，以上虚线代表数据流的方向。经过节点运算后，转换为不同的张量（数据）。

图 3-3 是静态的，不太容易看到数据流动的情况，可以访问 TensorFlow 官方网站 (https://www.TensorFlow.org/images/tensors_Flowing.gif)，查看以动画显示的“计算图”数据的流动。

3.3 TensorFlow 程序设计模式

如图 3-4 所示，TensorFlow 程序设计模式的核心是“计算图”，可分为两部分：建立“计算图”与执行“计算图”。

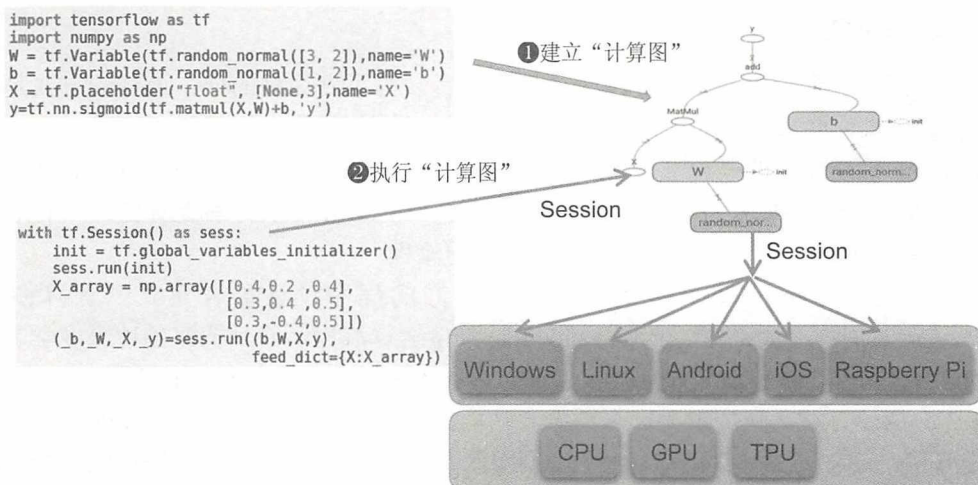


图 3-4

图 3-4 说明如下：

（1）建立“计算图”

我们可以使用 TensorFlow 提供的模块建立“计算图”。TensorFlow 提供的模块非常强大，我们可以设计张量运算流程，并且构建各种深度学习或机器学习模型。

（2）执行“计算图”

建立“计算图”后，我们就可以建立 Session 执行“计算图”了。在 TensorFlow 中，Session（原意为会话）的作用是在客户端和执行设备之间建立连接。有了这个连接，就可以将“计算图”在各种不同设备中执行，后续任何与设备之间的数据传输都必须通过 Session 来进行，并且最后获得执行后的结果。



3.4 Keras 介绍

Keras 是一个开放源码的高级深度学习程序库，使用 Python 编写，能够运行在 TensorFlow 或 Theano 之上。其主要作者和维护者是 Google 公司的工程师 FrançoisChollet，以 MIT 开放源码方式授权。

➤ 为何需要使用 Keras

Keras 使用最少的程序代码、花费最少的时间就可以建立深度学习模型，进行训练、评估准确率，并进行预测。

相对而言，使用 TensorFlow 这样低级的链接库虽然可以完全控制各种深度学习模型的细节，但是需要编写更多的程序代码，花费更多时间进行开发。

➤ Keras 的工作方式

Keras 是一个模型级（model-level）的深度学习链接库，Keras 只处理模型的建立、训练、预测等功能。深度学习底层的运行，例如张量（矩阵）运算，Keras 必须配合“后端引擎”（backend engine）进行运算。目前 Keras 提供了两种后端引擎：Theano 与 TensorFlow。

如图 3-5 所示，Keras 程序员只需要专注于建立模型，至于底层操作细节，例如张量（矩阵）运算，Keras 会帮你转化为 Theano 或 TensorFlow 相对指令。

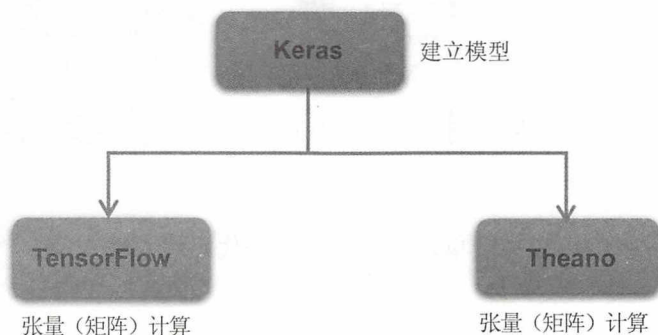


图 3-5

本书介绍的 Keras 范例都是使用 TensorFlow 作为后端引擎的，因为是以 TensorFlow 作为底层，所以之前章节中所介绍的 TensorFlow 的好处（例如跨平台与执行性能）就都具备了。

➤ Keras 深度学习链接库特色

- 简单快速地建立原型 prototyping: Keras 具备友好的用户界面、模块化设计、可扩充性。
- 已经内建各种类神经网络层级，例如卷积层 CNN、RNN，可以帮助我们快速建立神经网络模型。
- 通过后端引擎 Theano 与 TensorFlow，可以在 CPU 与 GPU 上运行。

- 以 Keras 开发的程序代码更简洁、可读性更高、更容易维护、更具生产力。
- Keras 的说明文件非常齐全，官方网站上提供的范例也非常浅显易懂。

3.5 Keras 程序设计模式

英文成语“piece of cake”的意思是“非常容易的事”，其实 Keras 的程序设计模式建立一个深度学习模型很简单，就好像做一个多层蛋糕。首先，建立一个蛋糕架。然后，我们不需要自己做每一层蛋糕，可以选择现成的蛋糕层，例如水果蛋糕层、巧克力蛋糕层等。我们可以指定每一层的“内容”，例如指定装饰水果种类与数量。只需要将每一层蛋糕加入蛋糕架上即可。最后就可以做出一个好吃又美观的多层蛋糕。

如图 3-6 所示，我们将建立多层感知器（Multilayer Perceptron）模型，输入层（ x ）共有 784 个神经元，隐藏层（ h ）共有 256 个神经元，输出层（ y ）共有 10 个神经元。建立这样的模型很简单，只需先建立一个蛋糕架，然后将神经网络层一层一层加上去即可。

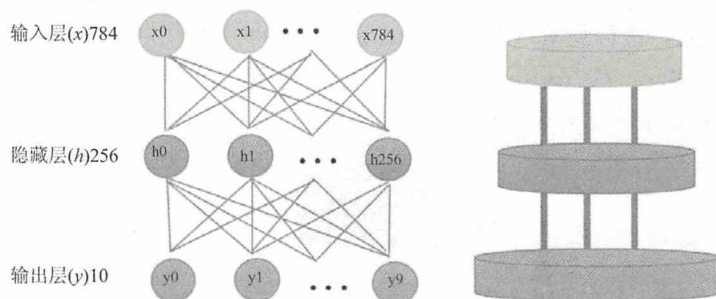


图 3-6

如图 3-7 所示，我们可以很简单地使用下列程序代码将“输入层”“隐藏层”与“输出层”加入模型中。

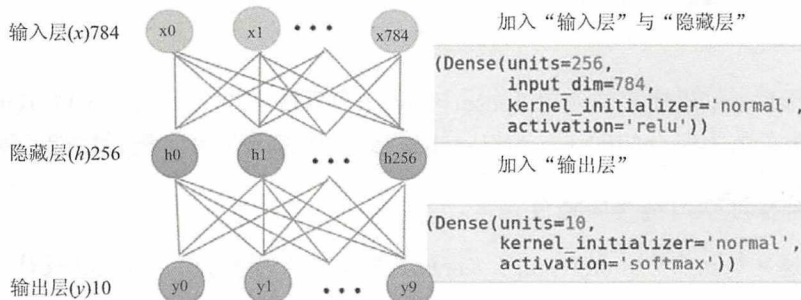


图 3-7

可以看出，在 Keras 建立多层感知器很简单。

1. 建立 Sequential 模型

Sequential 模型是多个神经网络层的线性堆叠。我们可以想象 Sequential 模型是一个蛋糕架，接下来可以加入一层一层的蛋糕。

```
model = Sequential()
```

2. 加入“输入层”与“隐藏层”到模型中

Keras 已经内建各种神经网络层（例如 Dense 层、Conv2d 层等），只要在之前建立的模型上加入我们选择的神经网络层就可以了（就好像在“蛋糕架”加入“蛋糕层”一样简单）。

以下程序代码加入“输入层”与“隐藏层”到模型中，就好像加入两层蛋糕。

```
model.add(Dense(units=256,
                 input_dim=784,
                 kernel_initializer='normal',
                 activation='relu'))
```

3. 加入“输出层”到模型

以下程序代码加入“输出层”到模型中，就好像再加入一层蛋糕。

```
model.add(Dense(units=10,
                 kernel_initializer='normal',
                 activation='softmax'))
```

以上基本就完成了多层感知器模型的建立，是不是很简单呢？这里只介绍了 Keras 程序设计的概念，详细程序代码的解说在后面章节会陆续介绍。

3.6 Keras 与 TensorFlow 比较

Keras 与 TensorFlow 比较见表 3-1。

表 3-1 Keras 与 TensorFlow 比较

	Keras	TensorFlow
学习难易度	简单	比较困难
使用弹性	中等	高
开发生产力	高	中等
执行性能	高	高
适合用户	初学者	高级用户
张量（矩阵）运算	不需要自行设计	需自行设计



► 轻松学会“深度学习”：先学 Keras 再学 TensorFlow

初学者学习 TensorFlow，就好像没有任何摄影经验的人一开始就使用单反相机且使用 M（手动）模式，必须学习一大堆有关光圈、快门等的知识，研究了老半天，还是没办法拍出一张像样的照片，会有很大的挫感。

初学者学习 Keras，就好像单反相机的初学者先以 Auto 自动模式来学习界面的构图等，这样就可以很容易地拍出一张张照片，再慢慢地使用 P 模式、A 模式、S 模式等，学习步骤自行设定，最后就可以使用 M 模式，完全掌控了。

本书希望能让初学者很轻松地学会“深度学习”，所以会先介绍 Keras 再介绍 TensorFlow。

因为大部分读者没有接触过深度学习模型，如果一开始就学习 TensorFlow，就要面对 TensorFlow 特殊的程序设计模式，并且还必须自行设计张量（矩阵）的运算，会有很大的挫折感。而先学习 Keras 可以让读者很容易地建立深度学习模型，并且训练模型，使用模型进行预测。等读者对深度学习模型有了一定认识后，再来学习 TensorFlow 就不会感觉那么困难了。

3.7 结论

本章我们介绍了 TensorFlow 与 Keras 的功能，并分别介绍了它们的程序设计模式。后续章节将介绍 TensorFlow 与 Keras 的安装，读者可以自行决定要使用 Windows 或 Linux 操作系统，我们将在第 4 章介绍 Windows 安装，在第 5 章介绍 Linux Ubuntu 安装。

第4章

在Windows中安装 TensorFlow与Keras

本章将介绍在 Windows 系统中安装 TensorFlow 与 Keras，并且启动 Jupyter Notebook 查看 TensorFlow 与 Keras 的版本。新版本的 TensorFlow 可以在 Windows 系统中安装，为用户带来很大的方便，毕竟大部分用户使用的都是 Windows 操作系统。

在 TensorFlow 官方网站介绍了很多安装 TensorFlow 的方式，网址如下：

https://www.tensorflow.org/versions/r0.10/get_started/os_setup

本书只介绍最简单的安装方式，就是以 Anaconda 安装。安装 TensorFlow 必须安装 Python。而安装 Python 最方便的方式就是使用软件包来安装。Anaconda 是一个 Python 发行版，其中包含大量的标准数学和科学计算软件包。安装 Anaconda 软件包时会同时帮我们安装很多软件包，包括 Jupyter Notebook、NumPy、SciPy、Matplotlib、Pandas 这 5 个用于数据分析、科学计算的常用软件包。

4.1 安装 Anaconda

安装 Anaconda 的步骤如下。

步骤01 下载 Anaconda 网址。

先打开浏览器，输入下列网址，显示出如图 4-1 所示的网页：

<https://www.continuum.io/downloads>



图 4-1

步骤02 运行下载后的 Anaconda，如图 4-2 所示。

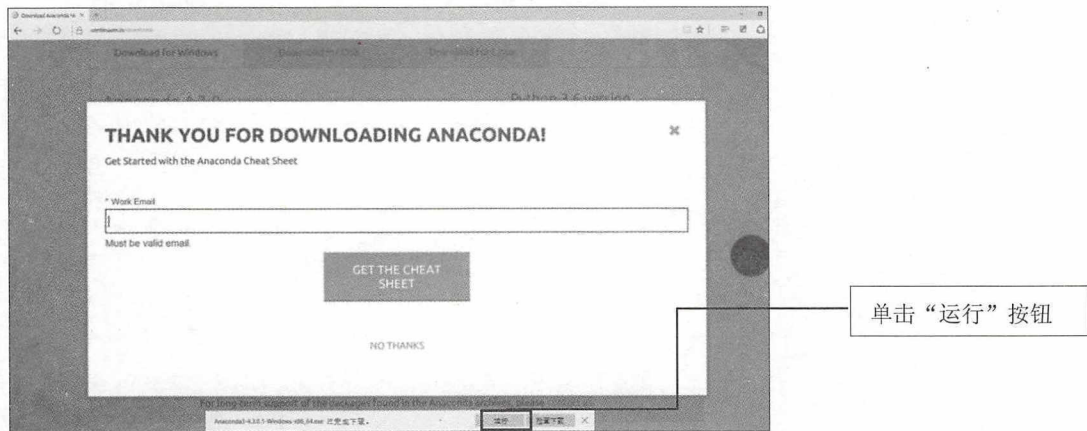


图 4-2

步骤03 单击 Next 按钮，如图 4-3 所示。

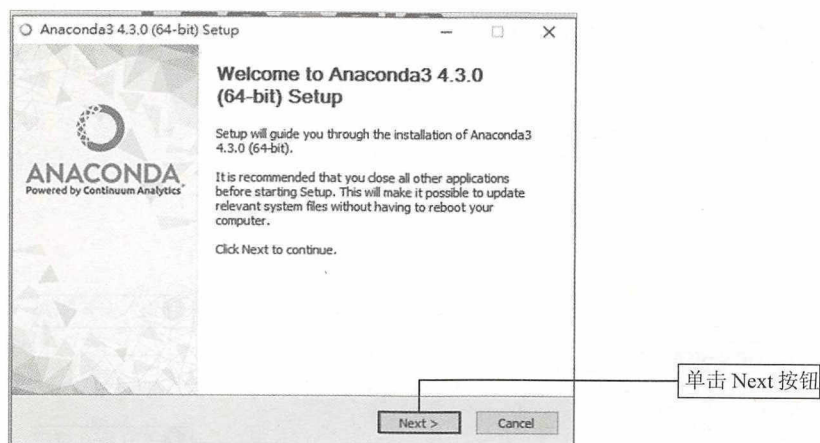


图 4-3

步骤04 单击 I Agree 按钮，如图 4-4 所示。

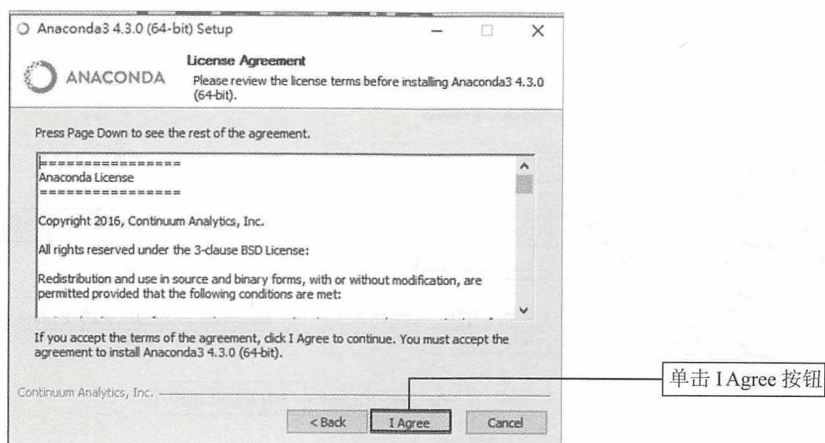


图 4-4

步骤05 单击 Next 按钮，如图 4-5 所示。

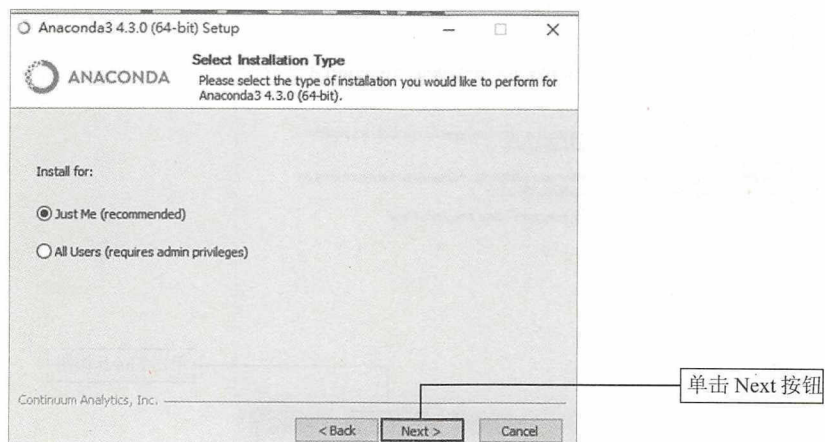


图 4-5

步骤06 设置安装目录，如图 4-6 所示。

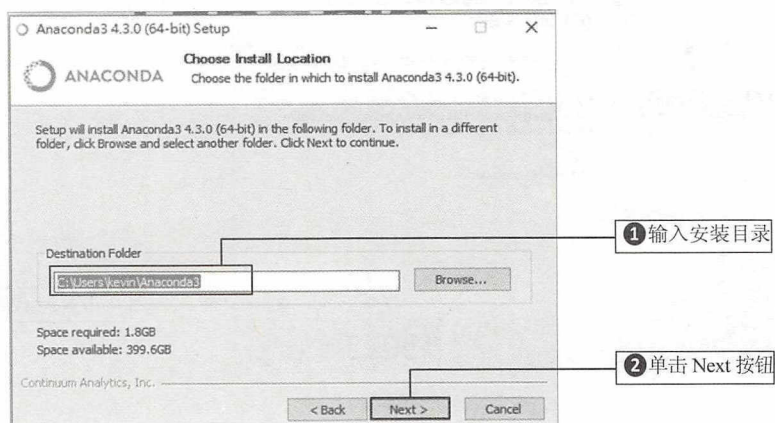


图 4-6

步骤07 设置 Anaconda，如图 4-7 所示。

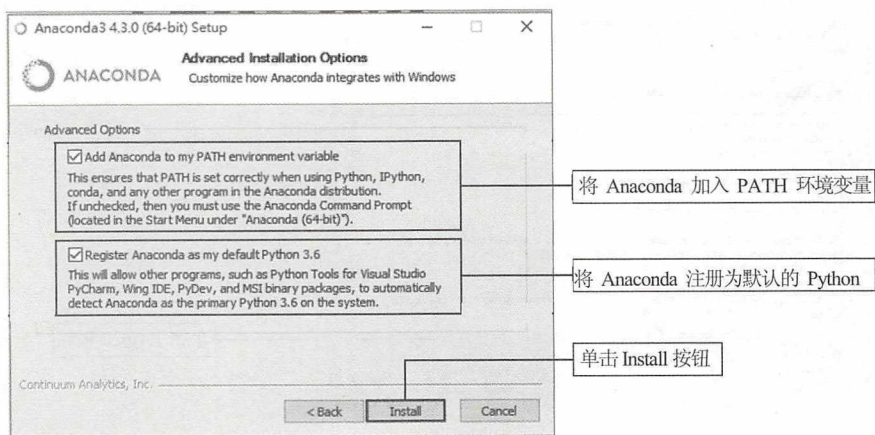


图 4-7

步骤08 安装完成，如图 4-8 所示。

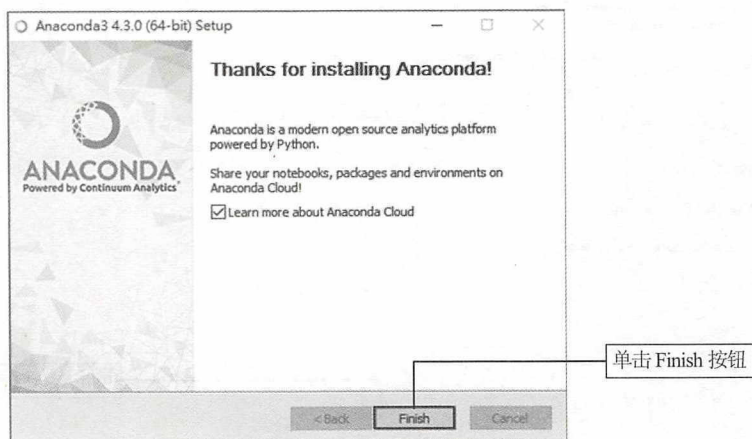


图 4-8

4.2 启动命令提示符

在 Linux 系统我们会使用“终端”程序输入命令，在 Windows 系统我们将使用“命令提示符”程序来输入命令。

步骤01 启动命令提示符程序，如图 4-9 所示。

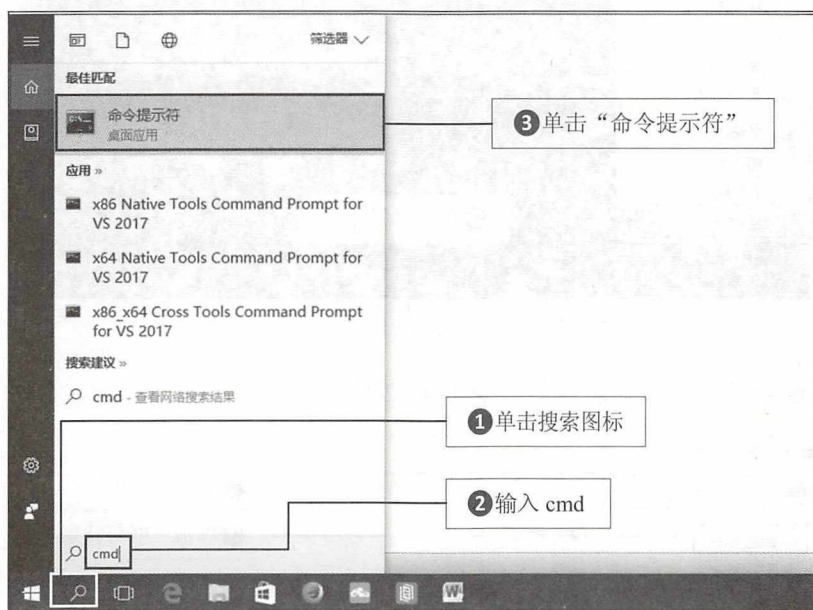


图 4-9

步骤02 打开命令提示符窗口，如图 4-10 所示。

在命令提示符窗口中可以输入命令。注意，命令提示符界面默认的输入法是微软的拼音输入法，按 Ctrl+【空格】组合键可切换为英文输入法，再输入命令。

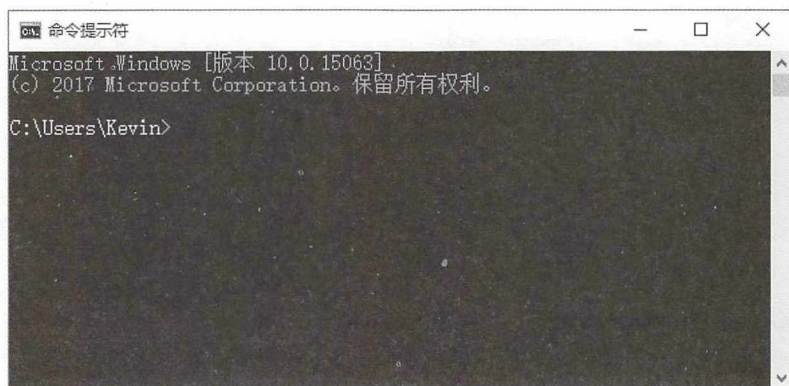


图 4-10

步骤03 设置命令提示符窗口，如图 4-11 所示。

因为计算机屏幕上默认的“黑底白字”在书上印刷看起来不清楚，所以后续会改为“白底黑字”。单击菜单图标，选择“属性”选项开始进行设置。

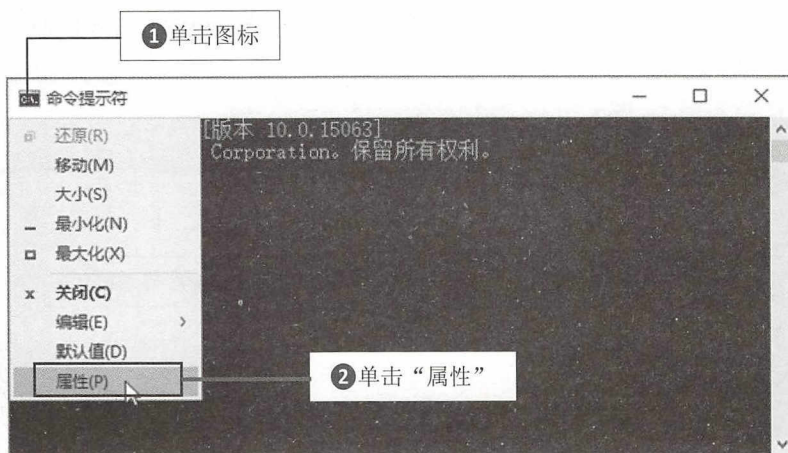


图 4-11

步骤04 设置命令提示符的背景颜色，如图 4-12 所示。

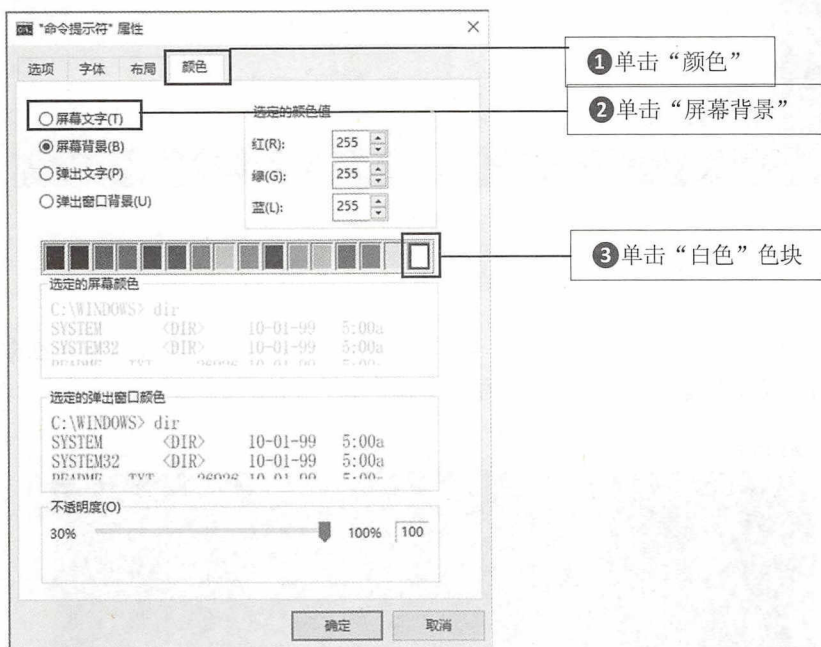


图 4-12

步骤05 设置命令提示符的文字颜色，如图 4-13 所示。

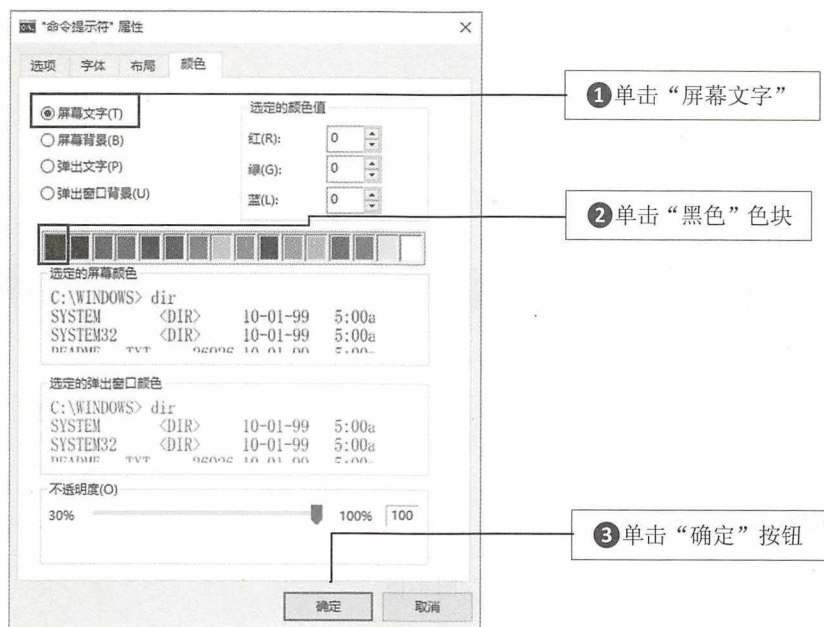


图 4-13

步骤06 设置完成后的命令提示符窗口，如图 4-14 所示。

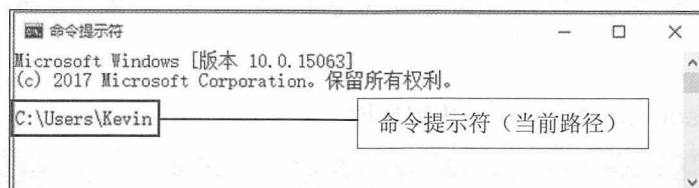


图 4-14

上面的屏幕显示界面出现后，命令提示符就是用户的目录。我们可以在此输入命令。

4.3 建立 TensorFlow 的 Anaconda 虚拟环境

为什么要使用 Anaconda 虚拟环境安装 TensorFlow？因为在一台计算机中，我们常常需要安装很多软件，但是每个软件所需要的 Python 的关联模块或版本都不相同。

例如，我们要使用 Python 开发网站系统，安装的网站框架可能需要 Python 3.x 的版本，但是要安装 TensorFlow 需要 Python 3.5 的版本，此时就会发生版本不一致的问题。为了解决这个问题，我们可以使用 Anaconda 虚拟环境来安装，让网站框架与 TensorFlow 分别安装在不同的虚拟环境中，这样就不会有版本不一致的问题了。

另外，本书会分别介绍如何使用 CPU 与 GPU 执行 TensorFlow 与 Keras。然而，CPU 与 GPU 所需要安装的 TensorFlow 版本不一样，所以我们会分别建立 CPU 与 GPU 的虚拟环境，

方便后续测试 CPU 与 GPU 的执行性能。

1. 建立工作目录

在命令提示符窗口输入下列命令：

➤ 建立并且切换到工作目录

```
md \pythonwork cd \pythonwork
```

执行后屏幕显示界面如图 4-15 所示。

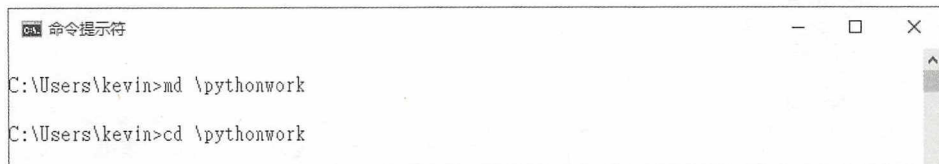


图 4-15

2. 建立 Anaconda 虚拟环境

下面使用 conda 命令建立一个新的 Python 3.5 Anaconda 虚拟环境，我们将虚拟环境命名为 TensorFlow。这个虚拟环境将用来安装 TensorFlow 的 CPU 版本。在命令提示符窗口输入下列命令：

➤ 建立 TensorFlow Anaconda 虚拟环境

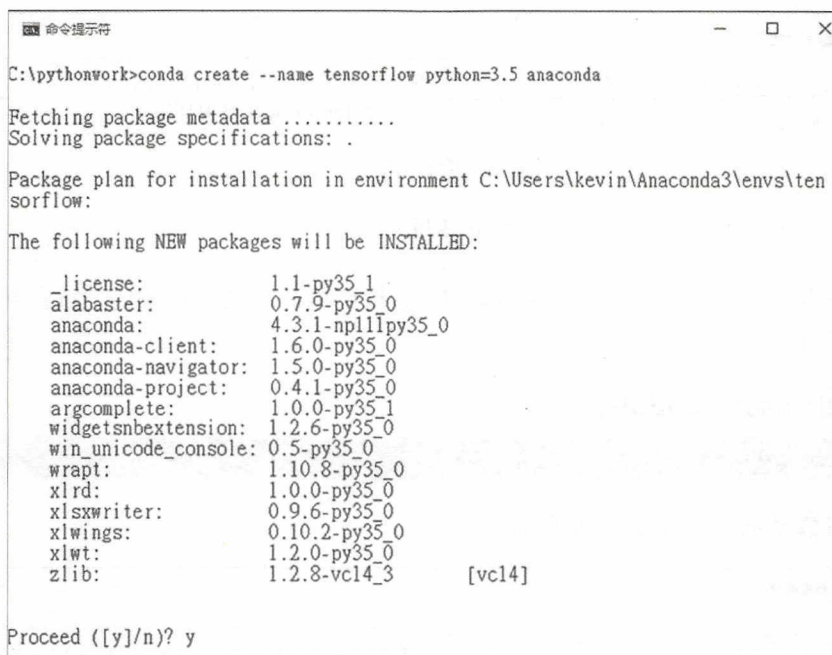
```
conda create --name tensorflow python=3.5 anaconda
```

以上命令说明见表 4-1。

表 4-1 命令说明

命令	说明
conda create	建立虚拟环境
--name tensorflow	虚拟环境的名称是 tensorflow
python=3.5	Python 版本是 3.5
anaconda	加入此命令选项，建立虚拟环境时，也会同时安装其他 Python 软件包，例如 Jupyter Notebook、NumPy、SciPy、Matplotlib、Pandas 这几个用于数据分析的软件包。如果没有加入此命令选项，就会建立一个空的虚拟环境，必须由用户自己再逐个安装其他软件包

执行后屏幕显示界面如图 4-16 所示。



```

C:\pythonwork>conda create --name tensorflow python=3.5 anaconda

Fetching package metadata .....
Solving package specifications: .

Package plan for installation in environment C:\Users\kevin\Anaconda3\envs\tensorflow:

The following NEW packages will be INSTALLED:

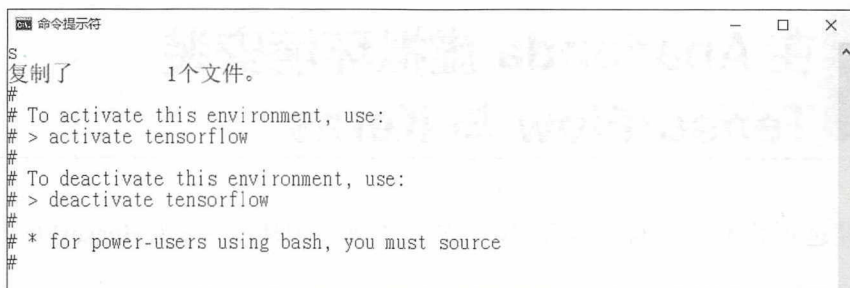
 _license:             1.1-py35_1
 alabaster:            0.7.9-py35_0
 anaconda:             4.3.1-np111py35_0
 anaconda-client:      1.6.0-py35_0
 anaconda-navigator:  1.5.0-py35_0
 anaconda-project:    0.4.1-py35_0
 argcomplete:         1.0.0-py35_1
 widgetsnbextension:  1.2.6-py35_0
 win_unicode_console: 0.5-py35_0
 wrapit:              1.10.8-py35_0
 xlrd:                1.0.0-py35_0
 xlswriter:           0.9.6-py35_0
 xlwings:             0.10.2-py35_0
 xlwt:               1.2.0-py35_0
 zlib:               1.2.8-vc14_3      [vc14]

Proceed ([y]/n)? y

```

图 4-16

按 Y 键之后, 就会开始安装 Anaconda 虚拟环境, 并且安装各个软件包。安装完成后屏幕显示界面如图 4-17 所示。



```

S
复制了      1个文件。
#
# To activate this environment, use:
# > activate tensorflow
#
# To deactivate this environment, use:
# > deactivate tensorflow
#
# * for power-users using bash, you must source
#

```

图 4-17

3. 启动 Anaconda 虚拟环境

建立 TensorFlow 的 Anaconda 虚拟环境后, 就可以启动这个虚拟环境了。在命令提示符窗口输入下列命令:

➤ 启动 Anaconda 虚拟环境

```
activate tensorflow
```

执行后屏幕显示界面如图 4-18 所示。

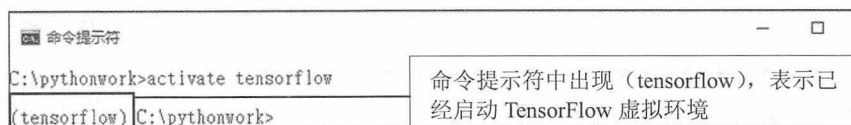


图 4-18

4. 关闭 TensorFlow 的 Anaconda 虚拟环境

当我们不再使用 TensorFlow 的 Anaconda 虚拟环境后，就可以关闭此虚拟环境。可在命令提示符窗口输入下列命令：

➤ 关闭 Anaconda 虚拟环境

```
deactivate tensorflow
```

执行后屏幕显示界面如图 4-19 所示。

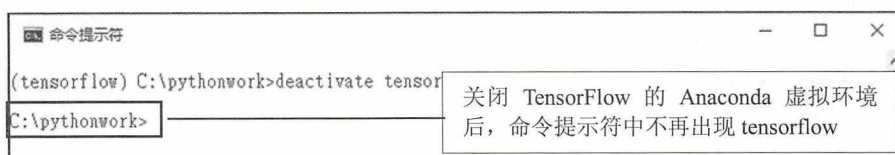


图 4-19

4.4 在 Anaconda 虚拟环境安装 TensorFlow 与 Keras

之前已经建立了 Anaconda 虚拟环境，现在要在此虚拟环境中安装 TensorFlow 与 Keras。

1. 启动 Anaconda 虚拟环境

安装 TensorFlow 与 Keras 前，先启动 TensorFlow 的 Anaconda 虚拟环境。在命令提示符窗口输入下列命令：

➤ 启动 Anaconda 虚拟环境

```
activate tensorflow
```

执行后屏幕显示界面如图 4-20 所示。

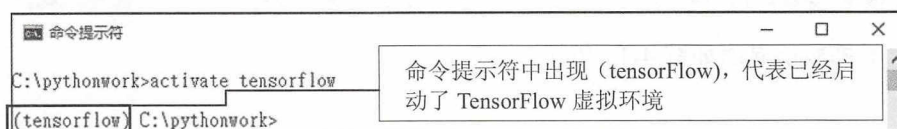


图 4-20



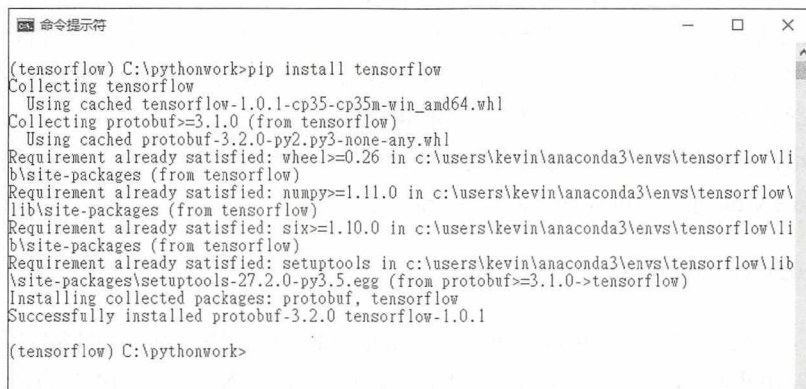
2. 安装 TensorFlow

在命令提示符窗口中输入下列命令：

➤ 安装 TensorFlow CPU 版本

```
pip install tensorflow
```

执行后屏幕显示界面如图 4-21 所示。



```
(tensorflow) C:\pythonwork>pip install tensorflow
Collecting tensorflow
  Using cached tensorflow-1.0.1-cp35-cp35m-win_amd64.whl
Collecting protobuf>=3.1.0 (from tensorflow)
  Using cached protobuf-3.2.0-py2.py3-none-any.whl
Requirement already satisfied: wheel>=0.26 in c:\users\kevin\anaconda3\envs\tensorflow\lib\site-packages (from tensorflow)
Requirement already satisfied: numpy>=1.11.0 in c:\users\kevin\anaconda3\envs\tensorflow\lib\site-packages (from tensorflow)
Requirement already satisfied: six>=1.10.0 in c:\users\kevin\anaconda3\envs\tensorflow\lib\site-packages (from tensorflow)
Requirement already satisfied: setuptools in c:\users\kevin\anaconda3\envs\tensorflow\lib\site-packages\setuptools-27.2.0-py3.5.egg (from protobuf>=3.1.0->tensorflow)
Installing collected packages: protobuf, tensorflow
Successfully installed protobuf-3.2.0 tensorflow-1.0.1

(tensorflow) C:\pythonwork>
```

图 4-21

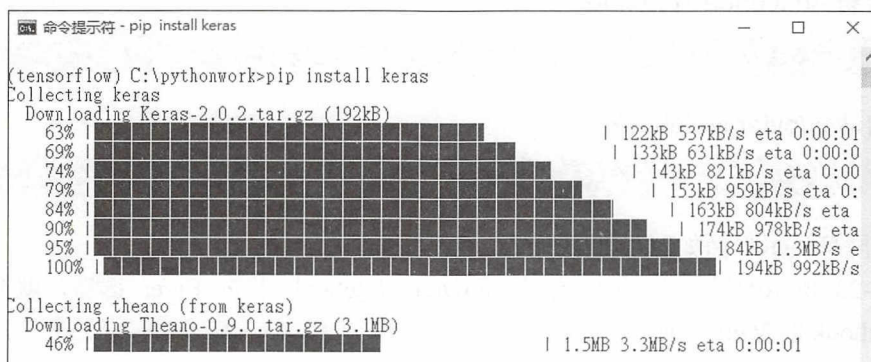
3. 安装 Keras

在命令提示符窗口输入下列命令：

➤ 安装 Keras

```
pip install keras
```

执行后屏幕显示界面如图 4-22 所示。



```
(tensorflow) C:\pythonwork>pip install keras
Collecting keras
  Downloading Keras-2.0.2.tar.gz (192kB)
    63% |#####| 122kB 537kB/s eta 0:00:01
    69% |#####| 133kB 631kB/s eta 0:00:0
    74% |#####| 143kB 821kB/s eta 0:00
    79% |#####| 153kB 959kB/s eta 0:
    84% |#####| 163kB 804kB/s eta
    90% |#####| 174kB 978kB/s eta
    95% |#####| 184kB 1.3MB/s e
    100% |#####| 194kB 992kB/s
Collecting theano (from keras)
  Downloading Theano-0.9.0.tar.gz (3.1MB)
    46% |#####| 1.5MB 3.3MB/s eta 0:00:01
```

图 4-22

4.5 启动 Jupyter Notebook

之前已经在 Anaconda 虚拟环境安装了 TensorFlow 与 Keras，现在我们要在 Jupyter Notebook 查看 TensorFlow 与 Keras 的版本。Jupyter Notebook 具备交互式界面，在 Web 界面输入 Python 命令后，可以立刻看到结果。我们还可以将数据分析的过程、执行后的命令与结果存储成笔记本。下次打开笔记本时，可以重新执行这些命令。Jupyter Notebook 笔记本可以包含文字、数学公式、程序代码、结果、图形和视频。

因为 Jupyter Notebook 是功能强大的交互式界面，所以在后续章节介绍的范例程序也会使用 Jupyter Notebook 示范 TensorFlow 与 Keras 指令。

1. 启动 Jupyter Notebook

在 4.3 节，当我们建立 TensorFlow 的 Anaconda 虚拟环境时，也同时安装了 Jupyter Notebook，所以不需要再安装，直接启动即可。启动 Jupyter Notebook 时先确认：

(1) 切换至工作目录，后续 Jupyter Notebook 读取与存盘都会在此工作目录。

(2) 确认已经启动 TensorFlow 的 Anaconda 虚拟环境，因为我们之前安装 TensorFlow 与 Keras 是在虚拟环境中，如果尚未启动这个虚拟环境就打开 Jupyter Notebook，那么执行 TensorFlow 与 Keras 程序时会出现 `ModuleNotFoundError` 错误。

在命令提示符窗口输入下列命令：

➤ 切换工作目录

```
cd \pythonwork
```

➤ 启动 Anaconda 虚拟环境

```
activate tensorflow
```

➤ 启动 Jupyter Notebook

```
jupyter notebook
```

执行后屏幕显示界面如图 4-23 所示。

在图 4-23 所示的运行界面中，输入 Jupyter Notebook 并按 Enter 键后，就会自动打开 Jupyter Notebook 的 Web 界面。

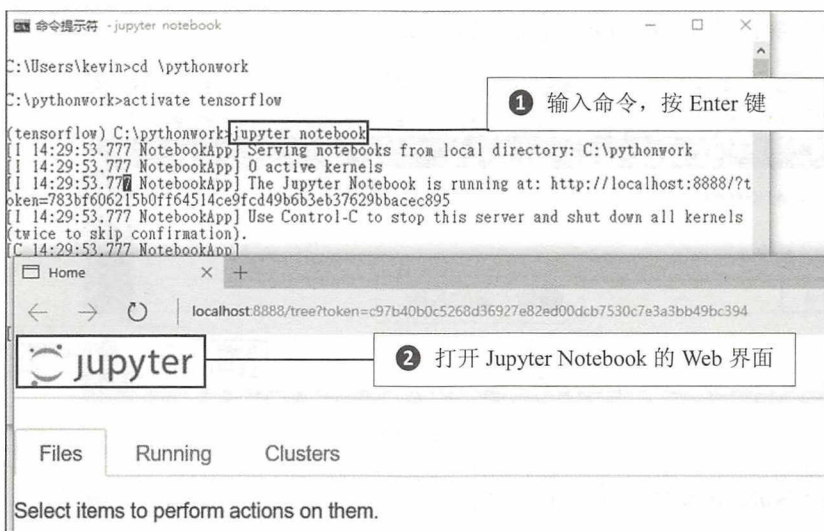


图 4-23

2. 建立新的 Notebook

进入 Jupyter Notebook 界面后，可以按照如图 4-24 所示的步骤新建 Notebook。

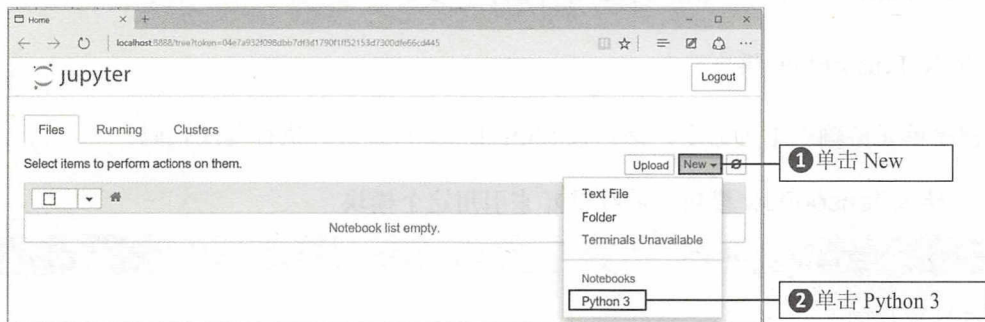


图 4-24

新建 Jupyter Notebook 后会打开浏览器新的页面，NoteBook 默认的名称是 Untitled，我们可以单击 Untitled 来修改 Notebook 的名称，如图 4-25 所示。

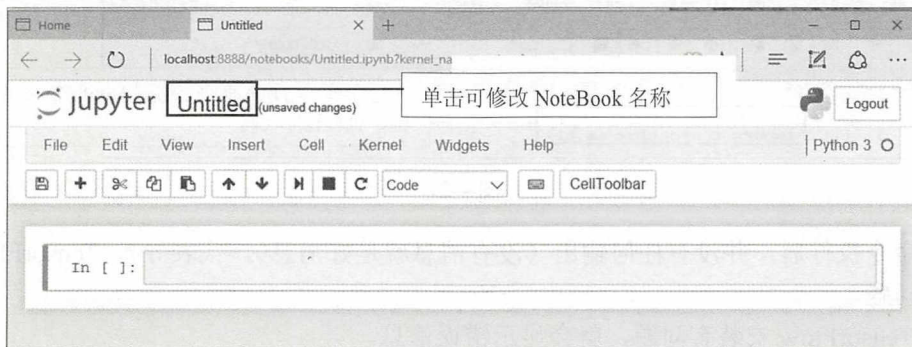


图 4-25

输入新的名称，然后单击 OK 按钮，如图 4-26 所示。

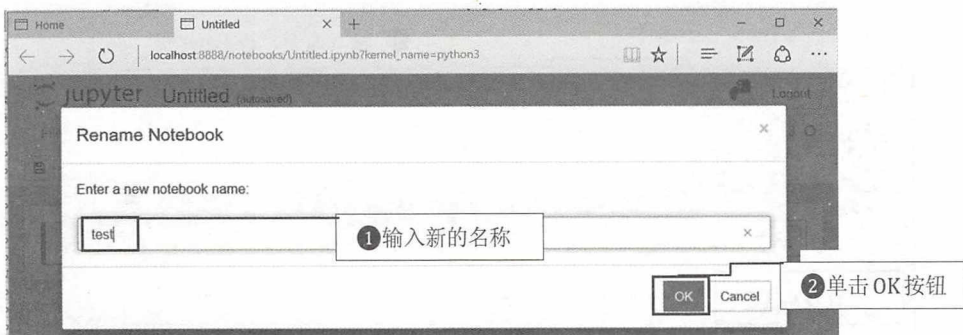


图 4-26

3. Jupyter Notebook 输入命令的方式

在 Jupyter Notebook 的 Cell（程序单元格）中输入程序代码，然后按 Shift+Enter 或 Ctrl+Enter 组合键来执行程序代码。这两种方式的差异如下。

- Shift+Enter: 执行后，光标会移到下一个程序单元格。
- Ctrl+Enter: 执行后，光标仍在当前的程序单元格。

4. 导入 TensorFlow 模块

在程序单元格输入下列命令，然后按 Shift+Enter 组合键，执行程序代码：

➤ 导入 TensorFlow 模块，后续以 tf 来引用这个模块

```
import tensorflow as tf
```

执行结果如图 4-27 所示。

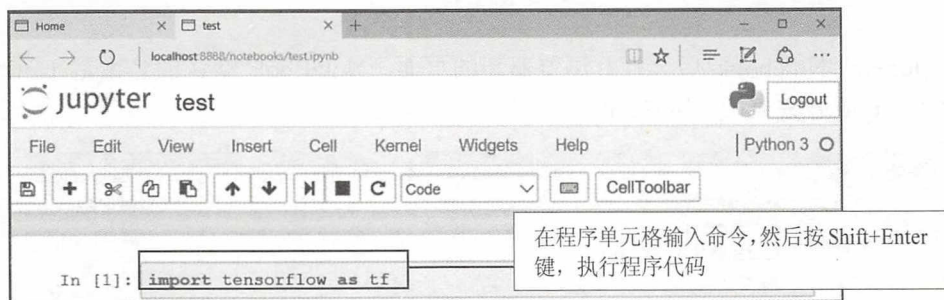


图 4-27

以上命令执行后，并没有任何输出（没有消息就是好消息），代表导入 TensorFlow 模块没有任何问题。

如果 TensorFlow 安装有问题，就会显示错误信息。



5. 查看 TensorFlow 版本

接下来，我们就可以查看 TensorFlow 版本了。在程序单元格输入下列命令：

➤ 查看 TensorFlow 版本

```
tf.__version__
```

执行结果如图 4-28 所示。

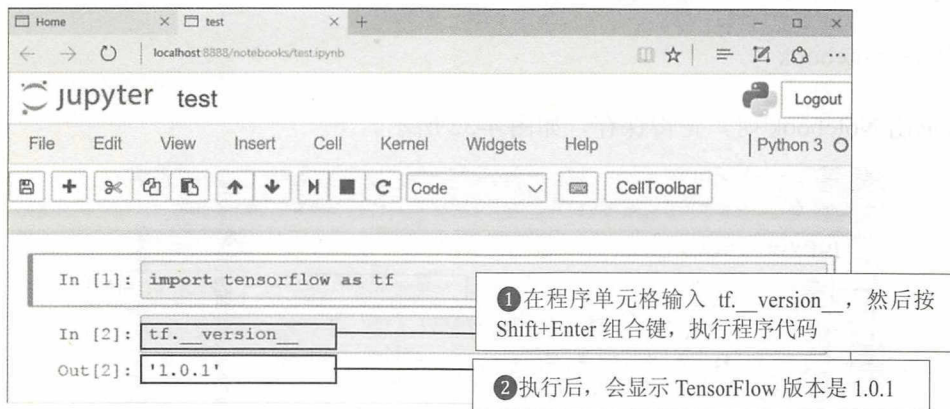


图 4-28

Jupyter Notebook 执行后屏幕显示界面如图 4-29 所示。

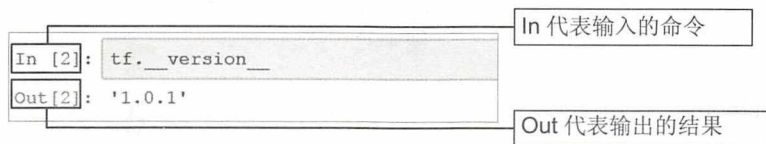


图 4-29

6. 导入 Keras 模块

在 Jupyter Notebook 的程序单元格输入程序代码。

➤ 导入 Keras 模块

```
import keras
```

执行结果如图 4-30 所示。

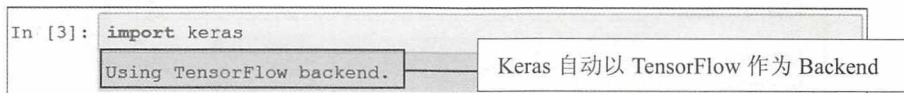


图 4-30

因为已经同时安装了 Keras 与 TensorFlow，所以导入 Keras 模块后，我们可以看到 Keras 自动以 TensorFlow 作为 Backend（后端）。

7. 查看 Keras 版本

在 Jupyter Notebook 的程序单元格输入如图 4-31 所示的程序代码，查看 Keras 版本。

```
In [5]: keras.__version__
Out[5]: '2.0.2'
```

图 4-31

以上执行结果显示 Keras 版本是 2.0.2。

8. 保存 Notebook

当要退出 Notebook 时，记得保存，如图 4-32 所示。

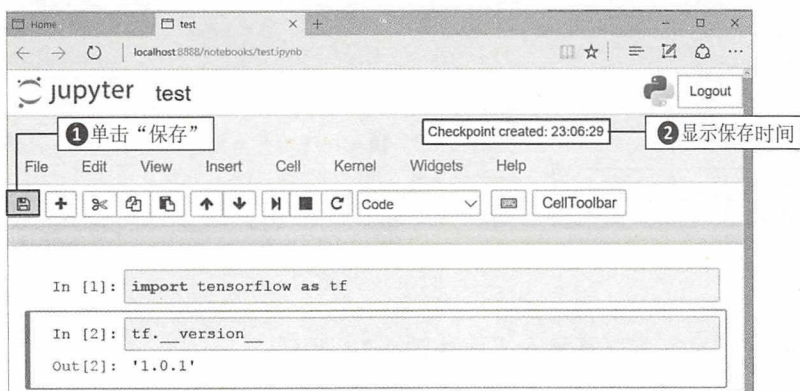


图 4-32

9. 关闭 Notebook

保存完成后，就可以关闭 Notebook 网页，步骤如图 4-33 所示。

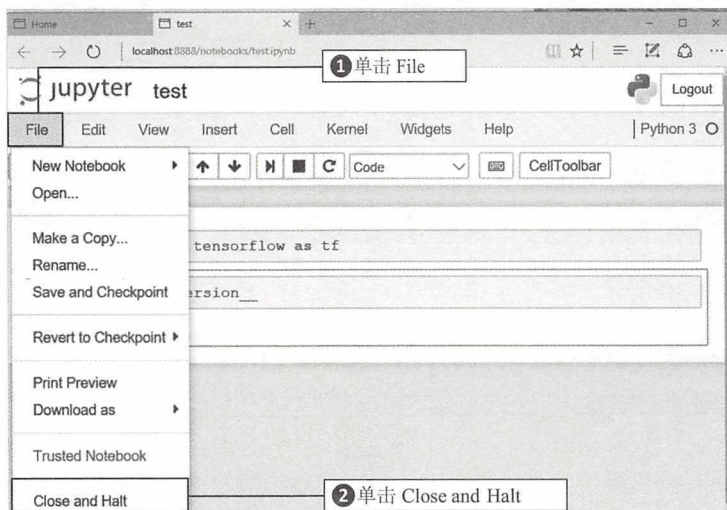


图 4-33



10. 打开之前保存的 Notebook

回到 Jupyter 网页，我们可以看到之前保存的 `test.ipynb`。如果要再次打开这个 Notebook，单击即可，如图 4-34 所示。

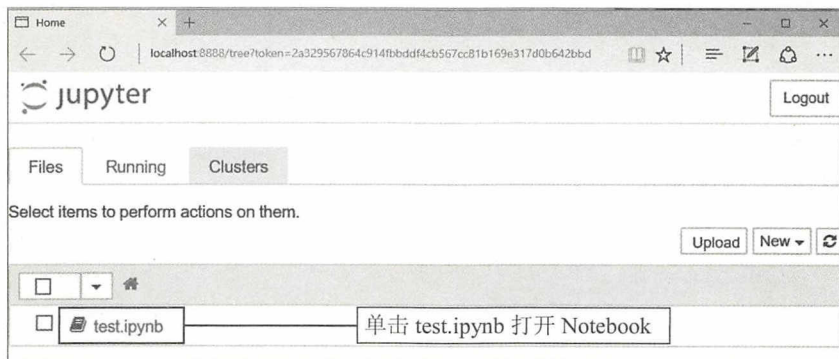


图 4-34

11. 关闭 Jupyter 浏览器（见图 4-35）



图 4-35

12. 关闭 Jupyter Notebook

关闭浏览器后，回到命令提示符窗口，按 `Ctrl+C` 组合键即可关闭 Jupyter Notebook 程序，如图 4-36 所示。

```

(tensorflow) C:\pythonwork>jupyter notebook
[I 23:14:11.413 NotebookApp] Serving notebooks from local directory: C:\pythonwork
[I 23:14:11.413 NotebookApp] 0 active kernels
[I 23:14:11.414 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/?token=9a20edfe6e855ed518975ea9bd2aaee8b589058247b71f57
[I 23:14:11.414 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 23:14:11.416 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
http://localhost:8888/?token=9a20edfe6e855ed518975ea9bd2aaee8b589058247b71f57
[I 23:14:11.951 NotebookApp] Accepting one-time-token-authenticated connection from ::1
[I 23:15:18.747 NotebookApp] Kernel started: 8cc69e24-17db-472d-91d1-2389f6ddf617
[I 23:15:57.445 NotebookApp] Kernel shutdown: 8cc69e24-17db-472d-91d1-2389f6ddf617
[I 23:22:15.664 NotebookApp] Interrupted...
[I 23:22:15.665 NotebookApp] Shutting down kernels
(tensorflow) C:\pythonwork>
  
```

图 4-36



4.6 结论

本章我们介绍了如何在 Windows 中安装 TensorFlow 与 Keras，并且介绍了如何打开 Jupyter Notebook 查看 TensorFlow 与 Keras 版本。我们将在第 5 章介绍如何在 Linux Ubuntu 中安装 TensorFlow 与 Keras。

第5章

在Linux Ubuntu中安装 TensorFlow与Keras

Linux 操作系统是大数据分析 with 机器学习很常用的平台。而 Ubuntu 是众多 Linux 操作系统版本中的一种，是一个开放源码、功能强大而且免费的操作系统。本书特别介绍在 Linux Ubuntu 操作系统中安装 TensorFlow 1.0+Keras 2.0。

本书假设读者已经安装了 Linux Ubuntu 操作系统，如果需要安装 Ubuntu 系统有关的资料，可以在网上找到详细的安装步骤，也可以参考笔者的另一本书《Python+Spark 2.0 + Hadoop 机器学习与大数据实战》，书中有详细的安装说明。

5.1 安装 Anaconda

安装 Anaconda 的步骤如下。

步骤01 复制安装 Anaconda 的下载网址。

在浏览器输入下列网址。

➤ 连接到 continuum 网址

<https://www.continuum.io/downloads>

向下浏览，我们可以看到 Anaconda for Linux，按照图 5-1 所示的步骤进行操作。



图 5-1

步骤02 下载 Anaconda3-4.2.0-Linux-x86_64.sh。

先在“终端”程序输入 wget 再按【空格】键，然后按 Ctrl+Shift+V 组合键来粘贴之前所复制的网址，如图 5-2 所示。

```
wget https://repo.continuum.io/archive/Anaconda3-4.2.0-Linux-x86_64.sh
```

```

user@Ubuntu1604: ~
user@Ubuntu1604:~$ wget https://repo.continuum.io/archive/Anaconda3-4.2.0-Linux-
x86_64.sh
--2017-01-17 09:49:48-- https://repo.continuum.io/archive/Anaconda3-4.2.0-Linux-
x86_64.sh
正在解析主机 repo.continuum.io (repo.continuum.io)... 104.16.19.10, 104.16.18.10
, 2400:cb00:2048:1::6810:120a, ...
正在连接 repo.continuum.io (repo.continuum.io)[104.16.19.10]:443... 已连接。
已发出 HTTP 请求, 正在等待响应... 200 OK
长度: 478051940 (456M) [application/octet-stream]
Saving to: 'Anaconda3-4.2.0-Linux-x86_64.sh'

Anaconda3-4.2.0-Lin 100%[=====] 455.91M 4.60MB/s in 1m 53s

2017-01-17 09:51:41 (4.02 MB/s) - 'Anaconda3-4.2.0-Linux-x86_64.sh' saved [47805
1940/478051940]

```

图 5-2

步骤03 安装 Anaconda。

在“终端”程序输入下列命令, 执行 Anaconda3-4.2.0-Linux-x86_64.sh。

```
bash Anaconda3-4.2.0-Linux-x86_64.sh -b
```

执行后屏幕显示界面如图 5-3 所示。

```

user@Ubuntu1604: ~
user@Ubuntu1604:~$ bash Anaconda3-4.2.0-Linux-x86_64.sh -b
PREFIX=/home/user/anaconda3
installing: python-3.5.2-0 ...
installing: _license-1.1-py35_1 ...
installing: _nb_ext_conf-0.3.0-py35_0 ...
installing: alabaster-0.7.9-py35_0 ...
installing: anaconda-clean-1.0.0-py35_0 ...
installing: anaconda-client-1.5.1-py35_0 ...
installing: anaconda-navigator-1.3.1-py35_0 ...
installing: argcomplete-1.0.0-py35_1 ...

```

图 5-3

以上指令加上“-b”是 batch 处理安装, 会自动省略。阅读 License 条款, 并自动安装到路径/home/hduser/anaconda2 下。

步骤04 编辑 ~/.bashrc 加入模块路径。

可以在“终端”程序中输入下列命令编辑 ~/.bashrc:

```
sudo gedit ~/.bashrc
```

打开编辑器屏幕显示界面, 输入如图 5-4 所示的内容, 输入完成后单击“保存”按钮。

*.bashrc (~) - gedit

打开(O) ▾
保存(S)

```
# added by Anaconda3 4.2.0 installer
export PATH="/home/user/anaconda3/bin:$PATH"
```

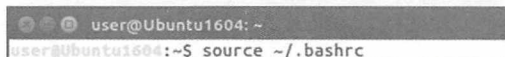
将 Anaconda 执行文件路径加入 PATH, 让我们在不同路径都可以执行 Anaconda

图 5-4

步骤05 使得 ~/.bashrc 的修改生效。

我们可以通过重新注销再登录，或使用下列命令让用户环境变量的设置生效（见图 5-5）：

```
source ~/.bashrc
```



```
user@Ubuntu1604: ~
user@Ubuntu1604:~$ source ~/.bashrc
```

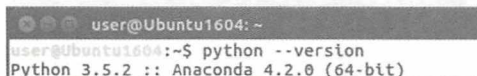
图 5-5

步骤06 查看 Python 版本。

可以在“终端”程序中输入下列命令：

```
python --version
```

执行后屏幕显示界面如图 5-6 所示，我们可以看到版本是 Anaconda 4.2.0。



```
user@Ubuntu1604: ~
user@Ubuntu1604:~$ python --version
Python 3.5.2 :: Anaconda 4.2.0 (64-bit)
```

图 5-6

5.2 安装 TensorFlow 与 Keras

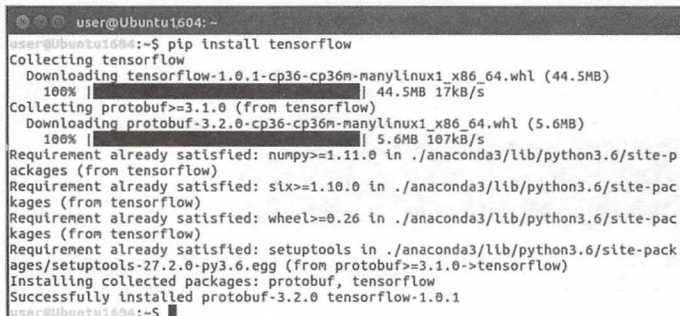
安装 TensorFlow 与 Keras 的步骤如下。

步骤01 安装 TensorFlow。

在“终端”程序中输入下列命令。

➤ 安装 TensorFlow（见图 5-7）

```
pip install tensorflow
```



```
user@Ubuntu1604: ~
user@Ubuntu1604:~$ pip install tensorflow
Collecting tensorflow
  Downloading tensorflow-1.0.1-cp36-cp36m-manylinux1_x86_64.whl (44.5MB)
    100% |#####| 44.5MB 17kB/s
Collecting protobuf>=3.1.0 (from tensorflow)
  Downloading protobuf-3.2.0-cp36-cp36m-manylinux1_x86_64.whl (5.6MB)
    100% |#####| 5.6MB 107kB/s
Requirement already satisfied: numpy>=1.11.0 in ./anaconda3/lib/python3.6/site-packages (from tensorflow)
Requirement already satisfied: six>=1.10.0 in ./anaconda3/lib/python3.6/site-packages (from tensorflow)
Requirement already satisfied: wheel>=0.26 in ./anaconda3/lib/python3.6/site-packages (from tensorflow)
Requirement already satisfied: setuptools in ./anaconda3/lib/python3.6/site-packages (from tensorflow)
Requirement already satisfied: setuptools in ./anaconda3/lib/python3.6/site-packages (from tensorflow)
Installing collected packages: protobuf, tensorflow
Successfully installed protobuf-3.2.0 tensorflow-1.0.1
user@Ubuntu1604:~$
```

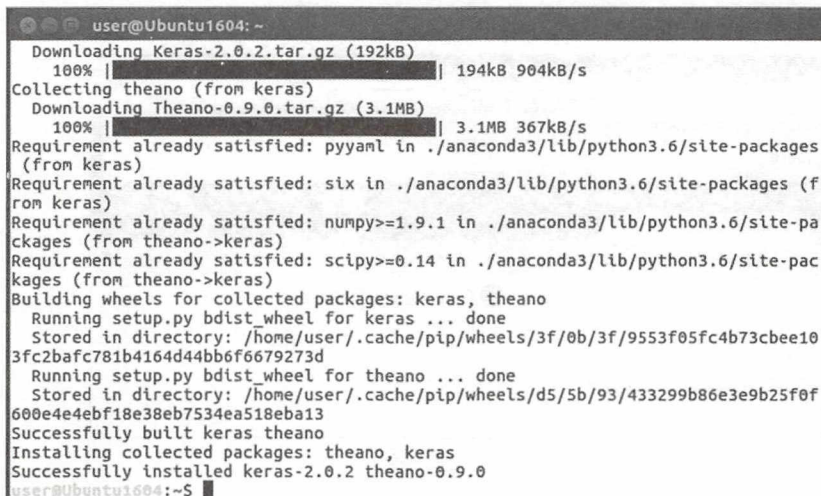
图 5-7

**步骤02** 安装 Keras。

在“终端”程序中输入下列命令。

➤ 安装 Keras (见图 5-8)

```
pip install keras
```



```
user@Ubuntu1604: ~  
Downloading Keras-2.0.2.tar.gz (192kB)  
100% |#####| 194kB 904kB/s  
Collecting theano (from keras)  
Downloading Theano-0.9.0.tar.gz (3.1MB)  
100% |#####| 3.1MB 367kB/s  
Requirement already satisfied: pyyaml in ./anaconda3/lib/python3.6/site-packages (from keras)  
Requirement already satisfied: six in ./anaconda3/lib/python3.6/site-packages (from keras)  
Requirement already satisfied: numpy>=1.9.1 in ./anaconda3/lib/python3.6/site-packages (from theano->keras)  
Requirement already satisfied: scipy>=0.14 in ./anaconda3/lib/python3.6/site-packages (from theano->keras)  
Building wheels for collected packages: keras, theano  
Running setup.py bdist_wheel for keras ... done  
Stored in directory: /home/user/.cache/pip/wheels/3f/0b/3f/9553f05fc4b73cbee103fc2bafc781b4164d44bb6f6679273d  
Running setup.py bdist_wheel for theano ... done  
Stored in directory: /home/user/.cache/pip/wheels/d5/5b/93/433299b86e3e9b25f0f600e4e4ebf18e38eb7534ea518eba13  
Successfully built keras theano  
Installing collected packages: theano, keras  
Successfully installed keras-2.0.2 theano-0.9.0  
user@Ubuntu1604:~$
```

图 5-8

5.3 启动 Jupyter Notebook

之前已经在 Ubuntu Linux 系统中安装了 TensorFlow 与 Keras,现在我们要在 Jupyter Notebook 中查看 TensorFlow 与 Keras 的版本。

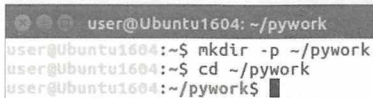
安装 Anaconda 时,也同时安装了 Jupyter Notebook,所以不需要再单独安装,直接启动即可。

步骤01 建立 ipynotebook 的工作目录。

可以在“终端”程序中输入下列命令,建立并切换到 ipynotebook 工作目录。

```
mkdir -p ~/pywork cd ~/pywork
```

按 Enter 键后,屏幕显示如图 5-9 所示。



```
user@Ubuntu1604: ~/pywork  
user@Ubuntu1604:~$ mkdir -p ~/pywork  
user@Ubuntu1604:~$ cd ~/pywork  
user@Ubuntu1604:~/pywork$
```

图 5-9

步骤02 进入 Jupyter Notebook。

进入工作目录后，在“终端”程序中输入下列命令，进入 Jupyter Notebook：

```
jupyter notebook
```

按 Enter 键后，就会启动浏览器，默认的网址是 <http://localhost:8888>，Jupyter Notebook 的界面如图 5-10 所示。



图 5-10

步骤03 启动 Jupyter Notebook。

进入 Jupyter Notebook 界面，如图 5-11 所示。关于 Jupyter Notebook 的简易使用说明，可参考 4.5 节的内容。

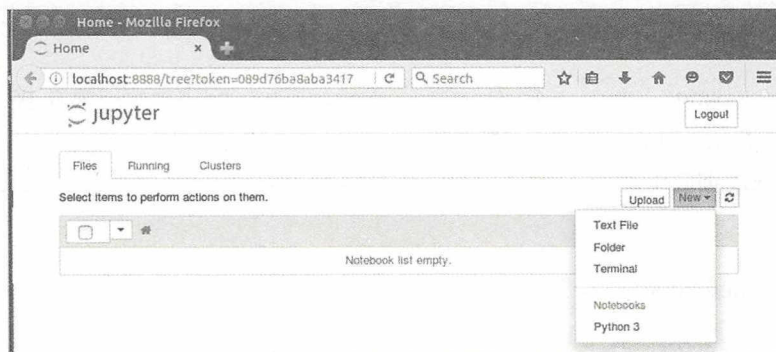


图 5-11

5.4 结论

本章介绍了如何在 Linux Ubuntu 中安装 TensorFlow 与 Keras，并且介绍了如何打开 Jupyter Notebook 来查看 TensorFlow 与 Keras 版本。从下一章开始，我们将开始使用 Keras 来设计程序。

第6章

Keras MNIST手写数字识别数据集

本章将介绍 MNIST 手写数字识别数据集，这是由 Yann LeCun 所搜集的，他也是 Convolution Neural Networks 的创始人。MNIST 数字文字识别数据集数据量不会太多，而且是单色的图像，比较简单，很适合深度学习的初学者用来练习建立模型、训练、预测。



MNIST 数据集共有训练数据 60 000 项、测试数据 10 000 项。MNIST 数据集中的每一项数据都由 images（数字图像）与 labels（真实的数字）所组成，如图 6-1 所示。

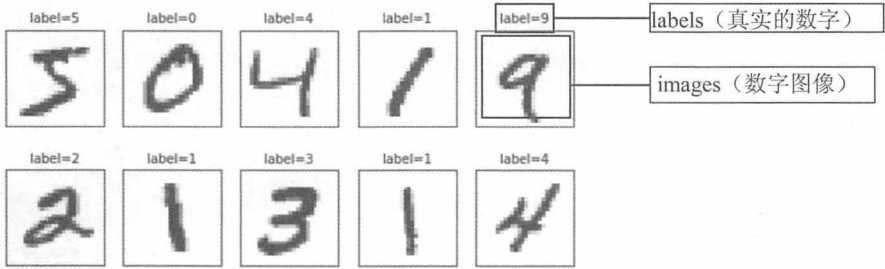


图 6-1

有关本章的完整程序代码参考范例程序 Keras_Mnist_Introduce.ipynb。范例程序的下载与安装参考本书附录 A。

6.1 下载 MNIST 数据

我们将创建以下 Keras 程序，下载并读取 MNIST 数据。

1. 导入 Keras 及相关模块

```
In [1]: import numpy as np
import pandas as pd
from keras.utils import np_utils
np.random.seed(10)
```

Using TensorFlow backend.

Keras 自动以 TensorFlow 作为 Backend

因为我们已经同时安装了 Keras 与 TensorFlow，所以导入 Keras 模块后，可以看到 Keras 自动以 TensorFlow 作为 Backend。

程序代码说明见表 6-1。

表 6-1 程序代码说明

程序代码	说明
<code>from keras.utils import np_utils</code>	导入 <code>keras.utils</code> ，因为后续要将 <code>label</code> 标签转换为 One-Hot Encoding（一位有效编码）
<code>import numpy as np</code>	导入 <code>numpy</code> 模块，NumPy 是 Python 语言的扩展链接库，支持维数组与矩阵运算
<code>np.random.seed(10)</code>	设置 <code>seed</code> 可以产生的随机数据

2. 导入 Keras 模块

Keras 已经提供了现成的模块，可以帮我们下载并读取 MNIST 数据，所以先导入

MNIST 模块。

```
In [2]: from keras.datasets import mnist
```

3. 第一次进行 MNIST 数据的下载

第一次执行 `mnist.load_data()` 方法，程序会检查用户目录下是否已经有 MNIST 数据集文件，如果还没有，就会下载文件。以下是第一次下载文件的屏幕显示界面，因为必须要下载文件，所以运行时间会比较长。

```
In [3]: (X_train_image, y_train_label), \
        (X_test_image, y_test_label) = mnist.load_data()

Downloading data from https://s3.amazonaws.com/img-datasets/mnist
.pkl.gz
```

4. 查看下载的 MNIST 数据文件

查看下载的 MNIST 数据文件，根据所使用的环境是 Windows 或 Linux Ubuntu 会有所不同，说明如下。

➤ 在 Windows 下查看已下载的 MNIST 数据文件

MNIST 数据文件下载后会存储在用户个人的文件夹中，因为笔者的用户名称是 kevin，所以下载后会存储在目录 `C:\Users\kevin\.keras\datasets` 中，文件名是 `mnist.npz`，如图 6-2 所示。我们也可以使用文件资源管理器来查看。

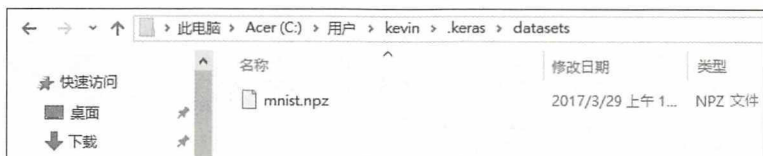


图 6-2

➤ 在 Linux Ubuntu 下查看已下载的 MNIST 数据文件

下载完成后，可以输入下列指令查看目录（见图 6-3）：

```
ll ~/.keras/datasets/mnist.pkl.gz
```

```
user@Ubuntu1604: ~
user@Ubuntu1604:~$
user@Ubuntu1604:~$ ll ~/.keras/datasets/mnist.pkl.gz
-rw-rw-r-- 1 user user 15296311 3月 11 10:44 /home/user/.keras/datasets/mnist.pkl.gz
```

图 6-3

5. 读取 MNIST 数据

当再次执行 `mnist.load_data()` 时，由于之前已经下载了文件，因此不需要再进行下载，只需要读取文件，这样运行速度就会快很多。

```
In [3]: (X_train_image, y_train_label), \
        (X_test_image, y_test_label) = mnist.load_data()
```

6. 查看 MNIST 数据

下载后，可以使用下列指令查看 MNIST 数据集的数据项数。

```
In [4]: print('train data=',len(x_train_image))
        print(' test data=',len(x_test_image))

train data= 60000
test data= 10000
```

从以上执行结果可知，数据分为两部分：

- train 训练数据 60 000 项。
- test 测试数据 10 000 项。

6.2 查看训练数据

先查看训练数据。

1. 训练数据是由 images 与 labels 所组成的

```
In [5]: print ('x_train_image:',x_train_image.shape)
        print ('y_train_label:',y_train_label.shape)

x_train_image: (60000, 28, 28)
y_train_label: (60000,)
```

images 与 labels 共 60000 项，images 是单色的数字图像，labels 是数字图像的真实值，如图 6-4 所示。

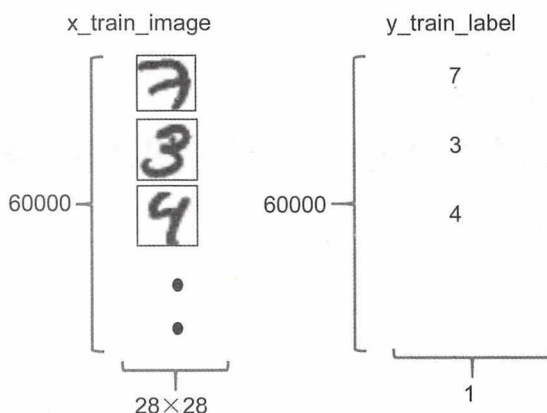


图 6-4



2. 定义 plot_image 函数显示数字图像

为了能够显示 images 数字图像，我们创建下列 plot_image 函数。

```
In [7]: import matplotlib.pyplot as plt
def plot_image(image):
    fig = plt.gcf()
    fig.set_size_inches(2, 2)
    plt.imshow(image, cmap='binary')
    plt.show()
```

以上程序代码说明见表 6-2。

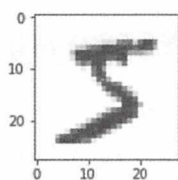
表 6-2 程序代码说明

程序代码	说明
import matplotlib.pyplot as plt	首先导入 matplotlib.pyplot 模块
def plot_image(image):	定义 plot_image 函数，传入 image 作为参数
fig = plt.gcf() fig.set_size_inches(2, 2)	设置显示图形的大小
plt.imshow(image, cmap='binary')	使用 plt.imshow 显示图形，传入参数 image 是 28×28 的图形，cmap 参数设置为 binary，以黑白灰度显示
plt.show()	开始绘图

3. 执行 plot_image 函数查看第 0 个数字图像

以下程序调用 plot_image 函数传入 mnist.train.images[0]，也就是训练数据集的第 0 项数据，从显示结果可以看到这是一个数字 5 的图形。

```
In [8]: plot_image(x_train_image[0])
```



4. 查看第 0 项 label 数据

```
In [8]: y_train_label[0]
Out[8]: 5
```

第 0 项 label 数据是第 0 个数字图像的真实值，所以是 5。

6.3 查看多项训练数据 images 与 label

接下来，我们将创建 `plot_images_labels_prediction` 函数，可以显示多项 MNIST 数据的 images 与 label。

1. 创建 `plot_images_labels_prediction()` 函数

我们希望能很方便地查看数字图形、真实的数字与预测结果，因此创建了下列函数。

```
In [13]: import matplotlib.pyplot as plt
def plot_images_labels_prediction(images, labels,
                                  prediction, idx, num=10):
    fig = plt.gcf()
    fig.set_size_inches(12, 14)
    if num>25: num=25
    for i in range(0, num):
        ax=plt.subplot(5,5, 1+i)
        ax.imshow(images[idx], cmap='binary')
        title= "label=" +str(labels[idx])
        if len(prediction)>0:
            title+=",predict="+str(prediction[idx])

        ax.set_title(title, fontsize=10)
        ax.set_xticks([]);ax.set_yticks([])
        idx+=1
    plt.show()
```

➤ 导入 pyplot 模块，后续会使用 plt 来引用

```
import matplotlib.pyplot as plt
```

➤ 定义 `plot_images_labels_prediction()` 函数

```
def plot_images_labels_prediction(images, labels, prediction, idx, num=10):
```

定义 `plot_images_labels_prediction()` 函数需要传入下列参数：

`images`（数字图像）、`label`（真实值）、`prediction`（预测结果）、`idx`（开始显示的数据 index）、`num`（要显示的数据项数，默认是 10，不超过 25）。

➤ 设置显示图形的大小

```
fig = plt.gcf() fig.set_size_inches(12, 14)
```

➤ 如果显示项数参数大于 25，就设置为 25，以免发生错误

```
if num>25: num=25
```

➤ for 循环执行程序块内的程序代码，画出 num 个数字图形

```
for i in range(0, num):
```




```
ax=plt.subplot(5,5, 1+i) # 建立 subgraph 子图形为 5 行 5 列
ax.imshow(images[idx], cmap='binary') # 画出 subgraph 子图形
title= "label="+str(labels[idx]) # 设置子图形 title, 显示标签字段
if len(prediction)>0: # 如果传入了预测结果
title+=" ,predict="+str(prediction[idx]) # 标题
title 加入预测结果 ax.set_title(title,fontsize=10) # 设置子图形的标题
title 与大小 ax.set_xticks([]);ax.set_yticks([]) # 设置不显示刻度
idx+=1 # 读取下一项
```

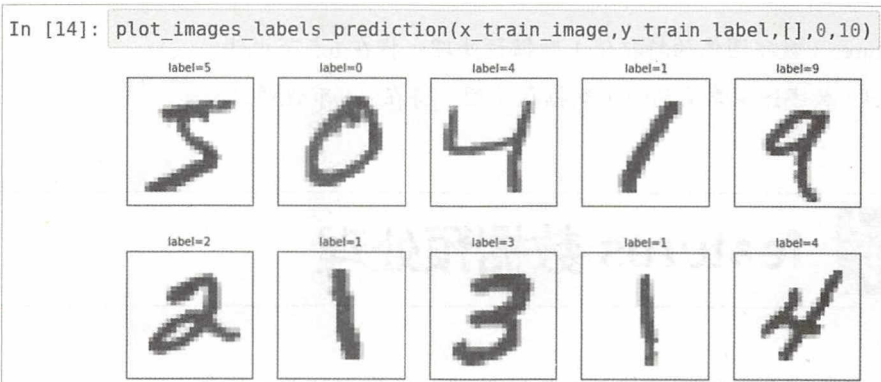
查看以上程序代码时可参考注释。

➤ 开始画图

```
plt.show()
```

2. 查看训练数据前 10 项数据

执行 `plot_images_labels_prediction()` 函数显示前 10 项训练数据。输入 `x_test_image` 和 `y_test_label`, 不过, 目前还没有预测结果 (`prediction`), 所以传入空 `list[]`, 从第 0 项数据开始一直显示到第 9 项数据。



3. 查看 test 测试数据

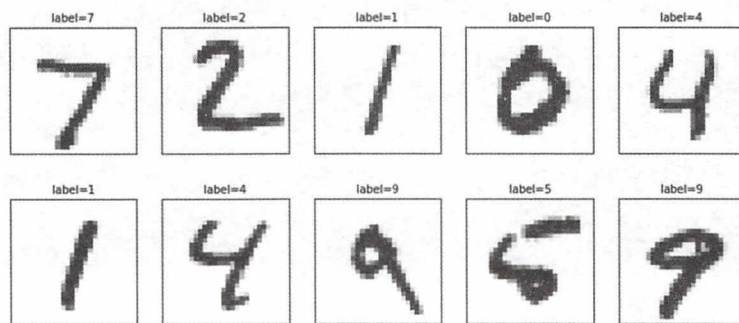
查看 test 测试数据项数, 我们可以看到共计 10000 项数据。

```
In [12]: print ('x_test_image:',x_test_image.shape)
print ('y_test_label:',y_test_label.shape)
x_test_image: (10000, 28, 28)
y_test_label: (10000,)
```

4. 显示 test 测试数据

执行 `plot_images_labels_prediction` 显示前 10 项测试数据。

```
In [15]: plot_images_labels_prediction(x_test_image,y_test_label,[],0,10)
```



6.4 多层感知器模型数据预处理

在下一章，我们将建立多层感知器模型（Multilayer Perceptron），必须先将 images 与 label 的内容进行预处理，才能使用多层感知器模型进行训练与预测。数据预处理分为以下两部分。

- features（数字图像的特征值）数据预处理：将在 6.5 节说明
- label（数字图像真实的值）数据预处理：将在 6.6 节说明。

6.5 features 数据预处理

features（数字图像的特征值）数据预处理可分为下列两个步骤：

- （1）将原本 28×28 的数字图像以 reshape 转换为一维的向量，其长度是 784，并且转换为 Float。
- （2）数字图像 image 的数字标准化。

步骤01 查看 image 的 shape。

可以用下列指令查看每一个数字图像的 shape 是 28×28。

```
In [14]: print ('x_train_image:',x_train_image.shape)
          print ('y_train_label:',y_train_label.shape)

x_train_image: (60000, 28, 28)
y_train_label: (60000,)
```

步骤02 将 image 以 reshape 转换。

下面的程序代码将原本 28×28 的二维数字图像以 reshape 转换为一维的向量，再以 astype 转换为 Float，共 784 个浮点数。

```
In [15]: x_Train = x_train_image.reshape(60000, 784).astype('float32')
         x_Test = x_test_image.reshape(10000, 784).astype('float32')
```

步骤03 查看转换为二维向量的 shape。

可以用下列指令查看每一个数字图像是 784 个浮点数。

```
In [16]: print ('x_train:',x_Train.shape)
         print ('x_test:',x_Test.shape)

         x_train: (60000, 784)
         x_test: (10000, 784)
```

步骤04 查看 images 图像的内容。

查看 images 第 0 项的内容。

```
In [16]: x_train_image[0]

         [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  3,  18, 18, 18, 126, 136, 175, 26, 166, 255, 247, 127,
          0,  0,  0,  0],
         [ 0,  0,  0,  0,  0,  0,  0,  0,  0, 30, 36, 94,
          154, 170, 253, 253, 253, 253, 225, 172, 253, 242, 195, 64,
          0,  0,  0,  0],
```

从以上执行结果可知，大部分都是 0，少部分是数字。每一个数字都是从 0 到 255 的值，代表图形每一个点灰度的深浅。

步骤05 将数字图像 images 的数字标准化。

images 的数字标准化可以提高后续训练模型的准确率，因为 images 的数字是从 0 到 255 的值，所以最简单的标准化方式是除以 255。

```
In [17]: x_Train_normalize = x_Train/ 255
         x_Test_normalize = x_Test/ 255
```

步骤06 查看数字图像 images 数字标准化后的结果。

使用下列指令查看数字图像 images 的数字标准化后的结果，都介于 0 与 1 之间。



```
In [18]: x_Train_normalize[0]
,      0.      , 0.      , 0.      , 0.      , 0.
,      0.      , 0.      , 0.01176471, 0.07058824,
0.07058824,
0.07058824, 0.49411765, 0.53333336, 0.68627453,
0.10196079,
0.65098041, 1.      , 0.96862745, 0.49803922, 0.
```

6.6 label 数据预处理

label（数字图像真实的值）标签字段原本是 0~9 的数字，必须以 One-Hot Encoding（一位有效编码）转换为 10 个 0 或 1 的组合，例如数字 7 经过 One-Hot Encoding 转换后是 0000000100，正好对应输出层的 10 个神经元。

步骤01 查看原本的 label 标签字段。

以下列指令来查看训练数据 label 标签字段的前 5 项训练数据，我们可以看到这是 0~9 的数字。

```
In [19]: y_train_label[:5]
Out[19]: array([5, 0, 4, 1, 9], dtype=uint8)
```

步骤02 label 标签字段进行 One-Hot Encoding 转换。

下面的程序代码使用 `np_utils.to_categorical` 分别传入参数 `y_train_label`（训练数据）与 `y_test_label`（测试数据）的 label 标签字段，进行 One-Hot Encoding 转换。

```
In [20]: y_TrainOneHot = np_utils.to_categorical(y_train_label)
y_TestOneHot = np_utils.to_categorical(y_test_label)
```

步骤03 查看进行 One-Hot Encoding 转换之后的 label 标签字段。

进行 One-Hot Encoding 转换之后，查看训练数据 label 标签字段的前 5 项数据，我们可以看到转换后的结果。

```
In [21]: y_TrainOneHot[:5]
Out[21]: array([[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
[ 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
[ 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

参考上面的结果，例如第 1 项数据，原来的真实值是 5，进行 One-Hot Encoding 转换后，只有第 5 个数字（由 0 算起）是 1，其余都是 0。



6.7 结论

在本章中，我们已经介绍了如何使用 Keras 下载并且读取 MNIST 数据集，并介绍了 MNIST 数据集的特色，也完成了数据的预处理。在下一章，我们可以使用 Keras 建立多层感知器模型进行训练，并且使用模型进行预测。

第7章

Keras多层感知器 识别手写数字

本章将介绍用 Keras 建立多层感知器模型，然后训练模型、评估模型的准确率，最后使用训练完成的模型识别 MNIST 手写数字。

关于多层感知器模型的详细介绍，可参考第2章。

有关本章的完整程序代码可参考范例程序 `Keras_Mnist_MLP_h256.ipynb`。范例程序的下载与安装可参考本书附录 A。

7.1 Keras 多元感知器识别 MNIST 手写数字图像的介绍

1. 多层感知器模型的介绍

为了能够识别 MNIST 手写数字图像，我们将建立如图 7-1 所示的多层感知器模型。

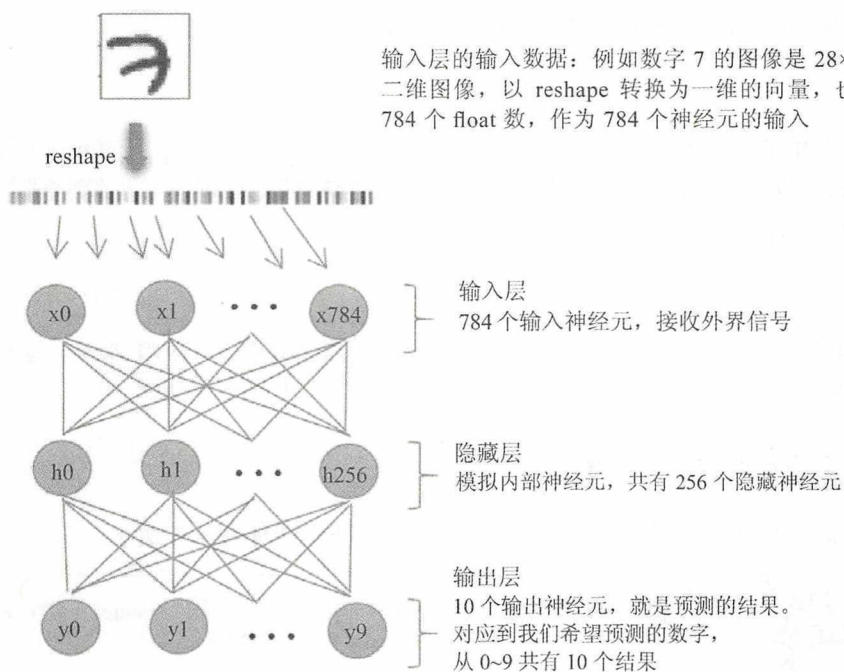


图 7-1

2. 多层感知器的训练与预测

建立如图 7-2 所示的多层感知器模型后，必须先训练模型才能够进行预测（识别）这些手写数字。

训练 (Training)



预测 (Predict)



图 7-2

以多层感知器模型识别 MNIST 数字图像可分为训练与预测。

➤ 训练

MNIST 数据集的训练数据共 60 000 项, 经过数据预处理后会产生 Features (数字图像特征值) 与 Label (数字真实的值), 然后输入多层感知器模型进行训练, 训练完成的模型就可以作为下一阶段预测使用。

➤ 预测

输入数字图像, 预处理后会产生 Features (数字图像特征值), 使用训练完成的多层感知器模型进行预测, 最后产生预测结果是 0~9 的数字。

3. 建立多层感知器模型的步骤

多层感知器识别 MNIST 数据集中的手写数字的步骤说明如图 7-3 所示。

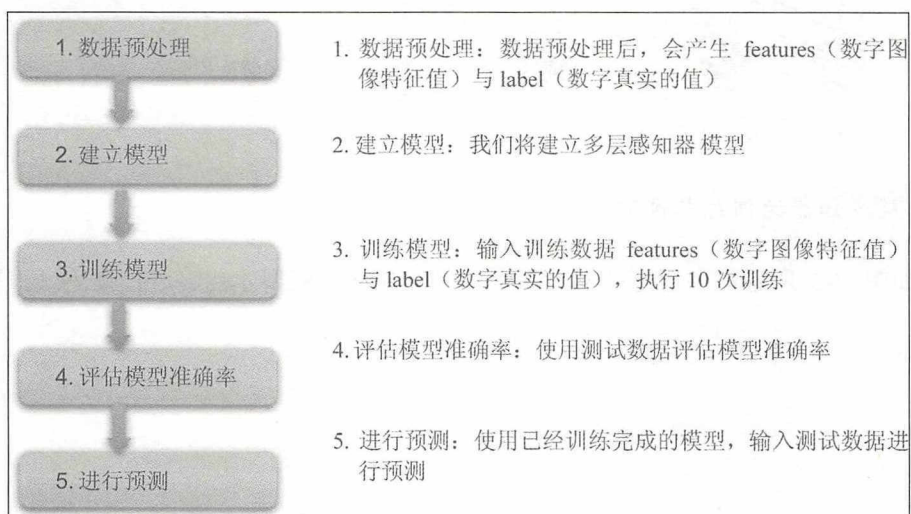


图 7-3

7.2 进行数据预处理

有关读取 MNIST 数据集数据并且进行数据预处理的详细介绍可参考第 6 章。

步骤01 导入所需模块。

```
In [1]: from keras.utils import np_utils
import numpy as np
np.random.seed(10)

Using TensorFlow backend.
```

步骤02 读取 MNIST 数据。

```
In [2]: from keras.datasets import mnist
(x_train_image,y_train_label),\
(x_test_image,y_test_label)= mnist.load_data()
```

步骤03 将 features（数字图像特征值）使用 reshape 转换。

下面的程序代码将原本 28×28 的数字图像以 reshape 转换成 784 个 Float 数。

```
In [3]: x_Train =x_train_image.reshape(60000, 784).astype('float32')
x_Test = x_test_image.reshape(10000, 784).astype('float32')
```

步骤04 将 features（数字图像特征值）标准化。

将 features（数字图像特征值）标准化可以提高模型预测的准确度，并且更快收敛。

```
In [4]: x_Train_normalize = x_Train / 255
x_Test_normalize = x_Test / 255
```

步骤05 label（数字真实的值）以 One-Hot Encoding 进行转换。

使用 np_utils.to_categorical 将训练数据与测试数据的 label 进行 One-Hot Encoding 转换。

```
In [5]: y_Train_OneHot = np_utils.to_categorical(y_train_label)
y_Test_OneHot = np_utils.to_categorical(y_test_label)
```

7.3 建立模型

我们将建立下列多层感知器模型，输入层（x）共有 784 个神经元，隐藏层（h）共有 256 个神经元，输出层（y）共有 10 个神经元，如图 7-4 所示。我们将使用下面的程序代码

建立多层感知器模型。

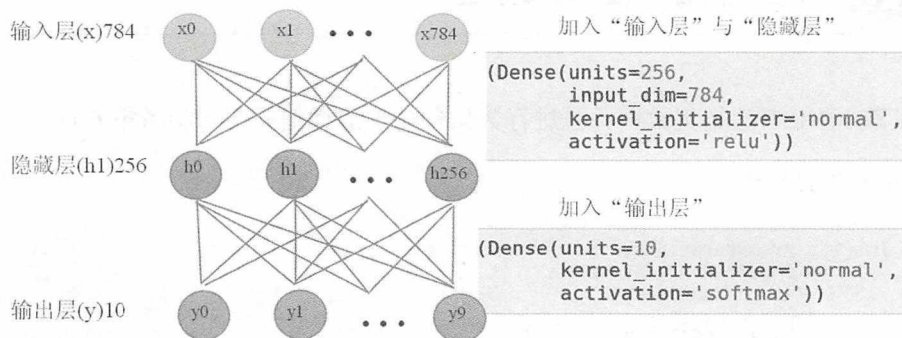


图 7-4

1. 导入所需模块

```
In [6]: from keras.models import Sequential
from keras.layers import Dense
```

2. 建立 Sequential 模型

建立一个线性堆叠模型，后续只需要使用 `model.add()` 方法将各个神经网络层加入模型即可。

```
In [7]: model = Sequential()
```

3. 建立“输入层”与“隐藏层”

以下程序代码将“输入层”与“隐藏层”加入模型，使用 `model.add` 方法加入 `Dense` 神经网络层。`Dense` 神经网络层的特色是：所有的上一层与下一层的神经元都完全连接。

```
In [8]: model.add(Dense(units=256,
input_dim=784,
kernel_initializer='normal',
activation='relu'))
```

建立 `Dense` 神经网络层需输入表 7-1 中的参数。

表 7-1 建立 `Dense` 神经网络层所需参数

参数	参数说明
<code>units=256</code>	定义“隐藏层”神经元个数为 256
<code>input_dim=784</code>	设置“输入层”神经元个数为 784（因为原本 28×28 的二维图像，以 <code>reshape</code> 转换为一维的向量，也就是 784 个 Float 数）
<code>kernel_initializer='normal'</code>	使用 <code>normal distribution</code> 正态分布的随机数来初始化 <code>weight</code> （权重）和 <code>bias</code> （偏差）
<code>activation</code>	定义激活函数为 <code>relu</code>

4. 建立“输出层”

使用下面的程序代码建立“输出层”，使用 `model.add` 方法加入 Dense 神经网络层，共有 10 个神经元，对应 0~9 十个数字。并且使用 `softmax` 激活函数进行转换，`softmax` 可以将神经元的输出转换为预测每一个数字的概率。

```
In [9]: model.add(Dense(units=10,
                        kernel_initializer='normal',
                        activation='softmax'))
```

建立“输出层”输入表 7-2 中的参数。

表 7-2 建立“输出层”所需参数

参数	参数说明
<code>units=10</code>	定义“输出层”神经元个数为 10
<code>kernel_initializer='normal'</code>	使用 normal distribution 正态分布的随机数来初始化 weight 与 bias
<code>activation</code>	定义激活函数为 softmax

以上建立 Dense 神经网络层不需要设置 `input_dim`，因为 Keras 会自动按照上一层的 `units` 是 256 个神经元，设置这一层的 `input_dim` 为 256 个神经元。

5. 查看模型的摘要

我们可以使用下列指令来查看模型的摘要。

```
In [10]: print(model.summary())
```

执行后屏幕显示界面如图 7-5 所示。

隐藏层共256个神经元			
Layer (type)	Output Shape	Param #	Connected to
dense_1 (Dense)	(None, 256)	200960	dense_input_1[0][0]
dense_2 (Dense)	(None, 10)	2570	dense_1[0][0]
Total params: 203,530			
Trainable params: 203,530			
Non-trainable params: 0			
None			
输出层共10个神经元			

图 7-5

我们可以看到共有以下两层。

- 隐藏层：共 256 个神经元，因为输入层与隐藏层是一起建立的，所以没有显示输入层。
- 输出层：共 10 个神经元。

6. 查看模型的摘要 Param

模型的摘要还有 Param 字段，说明如图 7-6 所示。

Param 计算方式 $784 \times 256 + 256 = 200960$		Param 计算方式 $256 \times 10 + 10 = 2570$	
Layer (type)	Output Shape	Param #	Connected to
dense_1 (Dense)	(None, 256)	200960	dense_input_1[0][0]
dense_2 (Dense)	(None, 10)	2570	dense_1[0][0]
Total params: 203,530			
Trainable params: 203,530		全部必须训练的 params 计算方式: $200960 + 2570 = 203530$	
Non-trainable params: 0			
None			

图 7-6

以上每一层 Param 都是超参数 (Hyper-Parameters)。我们需要通过反向传播算法更新神经元连接的权重与偏差。可参考第 2 章的公式。

建立输入层与隐藏层的公式如下：

```
h1 = relu(X×W1 + b1)
```

建立隐藏层与输出层的公式如下：

```
y = softmax(h1×W2 + b2)
```

所以每一层 Param 计算方式如下：

Param = (上一层神经元数量) × (本层的神经元数量) + (本层的神经元数量)

- 隐藏层的 Param 是 200 960，这是因为：

784 (输入层神经元数量) × 256 (隐藏层神经元数量) + 256 (隐藏层神经元数量) = $200\ 960$

- 输出层的 Param 是 2 570，这是因为：

256 (隐藏层神经元数量) × 10 (输出层神经元数量) + 10 (输出层神经元数量) = $2\ 570$

所以全部必须训练的超参数 (Trainable Params) 是每一层的 Param 的总和，计算方式如下：

200960 (隐藏层的 Param) + 2570 (输出层的 Param) = 203530

通常 Trainable Param 数值越大，代表此模型越复杂，需要更多时间进行训练。

7.4 进行训练

在我们建立好深度学习模型之后，就可以使用反向传播算法进行训练了，可参考第2章使用反向传播算法进行训练的说明。

1. 定义训练方式

在训练模型之前，我们必须使用 `compile` 方法对训练模型进行设置，指令如下：

```
In [11]: model.compile(loss='categorical_crossentropy',  
                        optimizer='adam', metrics=['accuracy'])
```

`compile` 方法需输入下列参数。

- **loss:** 设置损失函数，在深度学习中使用 `cross_entropy`（交叉熵）训练的效果比较好。
- **optimizer:** 设置训练时，在深度学习中使用 `adam` 优化器可以让训练更快收敛，并提高准确率。
- **metrics:** 设置评估模型的方式是准确率。

2. 开始训练

执行训练的程序代码如下：

```
In [12]: train_history =model.fit(x=x_Train_normalize,  
                                  y=y_Train_OneHot,validation_split=0.2,  
                                  epochs=10, batch_size=200,verbose=2)
```

以上程序代码说明如下：

使用 `model.fit` 进行训练，训练过程会存储在 `train_history` 变量中，需输入下列参数。

（1）输入训练数据参数

- `x=x_Train_normalize`（features 数字图像的特征值）。
- `y=y_Train_Onehot`（label 数字图像真实的值）。

（2）设置训练与验证数据比例

- 设置参数 `validation_split=0.2`。

训练之前 Keras 会自动将数据分成：80%作为训练数据，20%作为验证数据。因为全部数据是 60 000 项，所以分成：60 000×0.8=48 000 项作为训练数据，60 000×0.2=12 000 项作为验证数据。

(3) 设置 epoch (训练周期) 次数与每一批次项数

- epochs=10: 执行 10 个训练周期。
- batch_size=200: 每一批次 200 项数据。

(4) 设置显示训练过程

- verbose=2: 显示训练过程。

以上程序代码共执行了 10 次训练周期，每一次训练执行下列功能：

- 使用 48 000 项训练数据进行训练，分为每一批次 200 项，所以大约分为 240 个批次 (48 000/200=240) 进行训练。
- 训练完成后，会计算这个训练周期的准确率与误差，并且在 train_history 中新增一项数据记录。

➤ 以上程序代码执行后的结果如图 7-7 所示。

Train on 48000 samples, validate on 12000 samples				80% 作为训练数据, 20% 作为验证数据
Epoch 1/10	9s - loss: 0.0064 - acc: 0.9993 - val_loss: 0.0774 - val_acc: 0.9778			
Epoch 2/10	10s - loss: 0.0052 - acc: 0.9995 - val_loss: 0.0737 - val_acc: 0.9798			
Epoch 3/10	9s - loss: 0.0043 - acc: 0.9995 - val_loss: 0.0730 - val_acc: 0.9799			
Epoch 4/10	9s - loss: 0.0073 - acc: 0.9983 - val_loss: 0.0784 - val_acc: 0.9788			
Epoch 5/10	9s - loss: 0.0042 - acc: 0.9994 - val_loss: 0.0775 - val_acc: 0.9808			
Epoch 6/10	9s - loss: 0.0023 - acc: 0.9998 - val_loss: 0.0719 - val_acc: 0.9818			
Epoch 7/10	9s - loss: 0.0013 - acc: 1.0000 - val_loss: 0.0736 - val_acc: 0.9820			
Epoch 8/10	9s - loss: 9.1779e-04 - acc: 1.0000 - val_loss: 0.0727 - val_acc: 0.9822			
Epoch 9/10	9s - loss: 7.1102e-04 - acc: 1.0000 - val_loss: 0.0740 - val_acc: 0.9820			
Epoch 10/10	9s - loss: 5.8657e-04 - acc: 1.0000 - val_loss: 0.0742 - val_acc: 0.9826			
	使用训练数据计算误差与准确率		使用验证数据计算误差与准确率	

图 7-7

从以上执行结果可知，共执行了 10 个训练周期，并可以发现误差越来越小，准确率越来越高。

3. 建立 show_train_history 显示训练过程

之前的训练步骤会将每一个训练周期的准确率与误差记录在 train_history 变量中。我们可以使用下面的程序代码读取 train_history，以图表显示训练过程。

```
In [13]: import matplotlib.pyplot as plt
def show_train_history(train_history,train,validation):
    plt.plot(train_history.history[train])
    plt.plot(train_history.history[validation])
    plt.title('Train History')
    plt.ylabel(train)
    plt.xlabel('Epoch')
    plt.legend(['train', 'validation'], loc='upper left')
    plt.show()
```

程序代码说明如见表 7-3。

表 7-3 程序代码说明

程序代码	说明
%matplotlib inline	设置 matplotlib 在 jupyter note 网页内显示图形，如果少了这一指令，就会另开一个窗口显示图形
import matplotlib.pyplot as plt	导入 matplotlib.pyplot 模块，后续会使用 plt 来引用
def show_train_history(train_history, train, validation):	定义 show_train_history 函数，输入参数：之前训练过程所产生的 train_history 训练数据的执行结果 验证数据的执行结果
plt.title('Train History')	显示图的标题
plt.ylabel(train)	显示 y 轴的标签
plt.xlabel('Epoch')	设置 x 轴标签是'Epoch'
plt.legend(['train', 'validation'], loc='upper left')	设置图例是显示'train' 'validation'，位置在左上角

4. 画出准确率执行结果

下面的程序代码画出了准确率评估的执行结果，如图 7-8 所示。

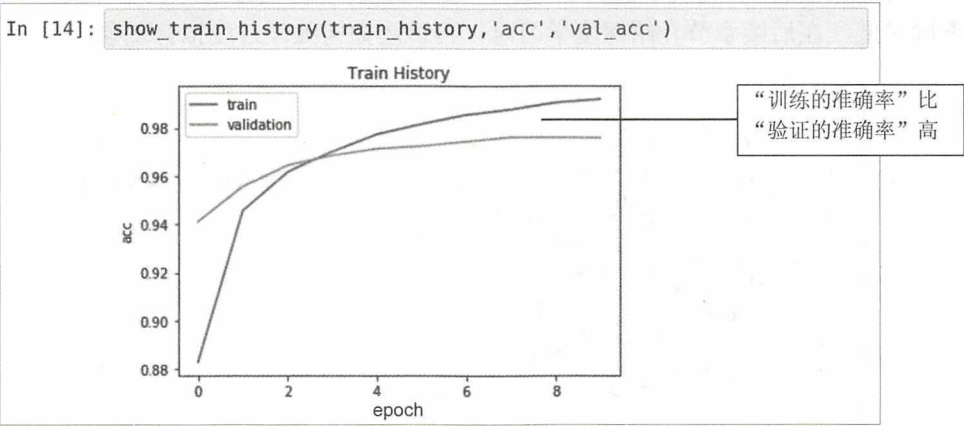


图 7-8

在以上执行后的屏幕显示界面中，“acc 训练的准确率”是蓝色（深色）的，“val_acc 验证的准确率”是黄色（浅色）的，（注意，本书是单色印刷，看不到蓝色和黄色，以深浅区

分)。以上共执行了 10 个训练周期，我们可以发现：

- 无论是训练还是验证，准确率都越来越高。
- 在 epoch 训练后期，“acc 训练的准确率”比“val_acc 验证的准确率”高。

➤ 为何“acc 训练的准确率”比“val_acc 验证的准确率”高？

这是因为计算准确率的数据不同。

- **acc 训练的准确率**：以训练的数据来计算准确率，因为相同的数据已经训练过了，又拿来计算准确率，所以准确率会比较高（就好像老师上课后，又使用上课的题目进行考试，准确率会比较高）。
- **val_acc 验证的准确率**：以验证数据来计算准确率，这些验证数据在之前训练时并未拿来训练，所以计算的准确率会比较低。但是，这样计算出来的准确率比较客观，比较符合真实情况（就好像老师上课后，使用独立的题库进行考试，这样学生考试的准确率没有那么高，但是比较客观反映学生的真实水平）。

如果“acc 训练的准确率”一直增加，但是“val_acc 验证的准确率”一直没有增加，就可能是过度拟合（overfitting）的现象。从以上的图形我们可以看到，“acc 训练的准确率”比“val_acc 验证的准确率”高，虽然差异不是很大，但仍有轻微过度拟合的现象。

➤ 过度拟合的现象

什么是过度拟合呢？

如图 7-9 所示，有两个分类圆形与星形，我们希望训练后找出一条线，可以将圆形与星形进行分类。实线是我们希望找到的最佳分类线，可是训练过程太久或范例太少会导致虚线过度适应训练数据中特化且随机的特征。其结果是虽然在训练时准确率高，但是使用未知数据时的准确率低。在后续章节介绍深度学习时，再讲述如何处理过度拟合现象。

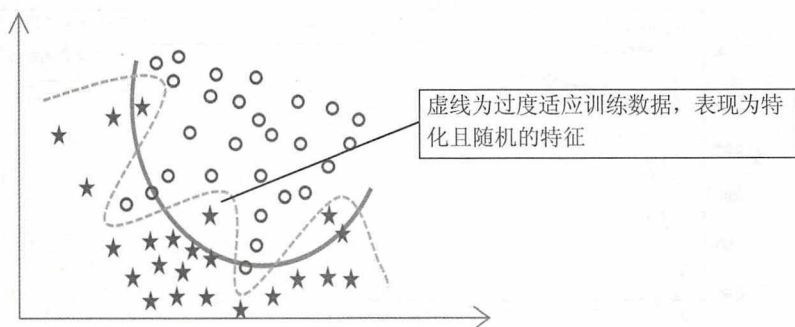


图 7-9

➤ 以测试数据评估模型准确率

在完成所有训练周期之后，在 7.4 节我们还会以测试数据评估模型准确率，这是另一组

独立的数据，所以计算准确率会更客观（就好像老师上了一个学期课之后，使用另一组独立的题库进行考试，就像期末考试一样，这样考试的结果才比较客观）。

5. 画出误差执行结果（见图 7-10）

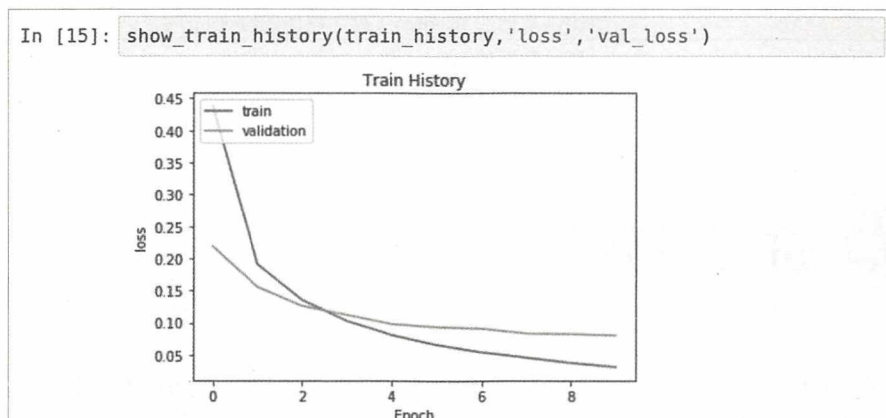


图 7-10

在上面执行结果的屏幕显示界面中，“loss 训练的误差”是深色的，“val_loss 验证的误差”是浅色的，总共执行了 10 个训练周期，我们可以发现：

- 无论是训练还是验证，验证的误差都越来越低。
- 在 Epoch 训练后期，“loss 训练的误差”比“val_loss 验证的误差”小。

7.5 以测试数据评估模型准确率

之前我们已经完成了训练，现在要使用 test 测试数据来评估模型准确率。

1. 评估模型准确率

下面的程序代码用于评估模型准确率。

```
In [16]: scores = model.evaluate(x_Test_normalize, y_Test_OneHot)
          print()
          print('accuracy=', scores[1])

          9856/10000 [=====>.] - ETA: 0s
          accuracy= 0.9759
```

以上程序代码的执行结果是准确率为 0.97。程序代码的说明见表 7-4。

表 7-4 程序代码说明

程序代码	说明
<code>scores = model.evaluate(</code>	使用 <code>model.evaluate</code> 评估模型的准确率，评估后的准确率会存储在 <code>scores</code> 中
<code>x= x_Test_normalize,</code>	测试数据的 <code>features</code> （数字图像的特征值）
<code>y= y_Test_Onehot)</code>	测试数据的 <code>label</code> （数字图像真实的值）
<code>print('accuracy=',scores[1])</code>	显示准确率

7.6 进行预测

通过之前的步骤，我们建立了模型，并且完成了模型训练，准确率达到还可以接受的 0.97，接下来我们将使用此模型进行预测。

1. 执行预测

我们可以用下列指令执行预测。

```
In [17]: prediction=model.predict_classes(x_Test)
          9632/10000 [=====>..] - ETA: 0s
```

上面的程序代码使用 `model.predict_classes` 输入参数 `x_Test`（测试数据的数字图像）进行预测，预测结果存储在 `prediction` 变量中。

2. 预测结果

我们可以用下列指令来查看预测结果的前 10 项数据。

```
In [18]: prediction
Out[18]: array([7, 2, 1, ..., 4, 5, 6], dtype=int64)
```

可以看到第 1 项预测的结果是 7，第 2 项预测是 2，诸如此类。

3. 显示 10 项预测结果

使用上一章创建的 `plot_images_labels_prediction` 函数显示预测结果，输入参数：`x_test_image`（测试数据图像）、`y_test_label`（测试数据真实的值）、`prediction`（预测结果）和 `idx=340`（显示第 340 到 349 共 10 项）。

```
In [20]: plot_images_labels_prediction(x_test_image,y_test_label,
          prediction,idx=340)
```

执行后预测结果如图 7-11 所示。

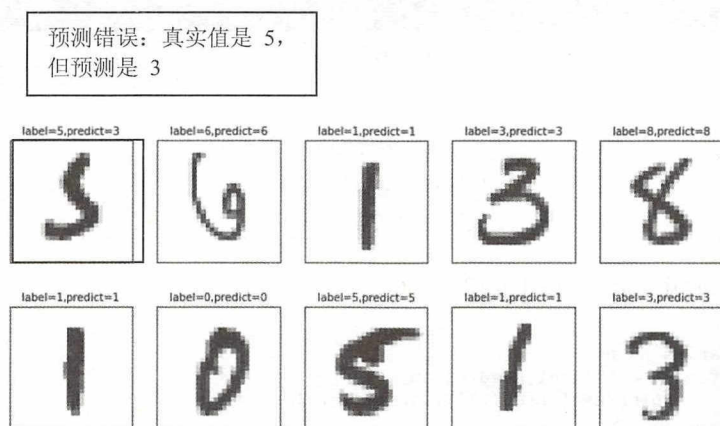


图 7-11

我们可以看到有一项预测错误：label（真实值）是 5，但 predict（预测值）是 3，这个手写数字图像确实挺潦草的，难怪会识别错误。

7.7 显示混淆矩阵

在上一节中我们看到了一个预测错误：真实值是 5，但是预测值是 3。如果我们想要进一步知道在所建立的模型中哪些数字的预测准确率最高，哪些数字最容易混淆（例如真实值是 5，但是预测值是 3），就可以使用混淆矩阵（confusion matrix）来显示。

在机器学习领域，特别是统计分类的问题，混淆矩阵也称为误差矩阵（error matrix），是一种特定的表格显示方式，可以让我们以可视化的方式了解有监督的学习算法的结果，看出算法模型是否混淆了两个类（将某一个标签预测成为另一个标签）。

1. 使用 pandas crosstab 建立混淆矩阵

Pandas 提供了建立混淆矩阵的功能。

```
In [21]: import pandas as pd
         pd.crosstab(y_test_label, prediction,
                     rownames=['label'], colnames=['predict'])
```

程序代码说明见表 7-5。

表 7-5 程序代码说明

程序代码	说明
<code>import pandas as pd</code>	导入 pandas 模块，后续会以 pd 来引用
<code>pd.crosstab(y_test_label, prediction, rownames=['label'], colnames=['predict'])</code>	使用 pd.crosstab 建立混淆矩阵，输入参数：测试数据数字图像的真实值 测试数据数字图像的预测结果 设置行的名称是 label 设置列的名称是 predict

执行后显示出混淆矩阵，如图 7-12 所示。

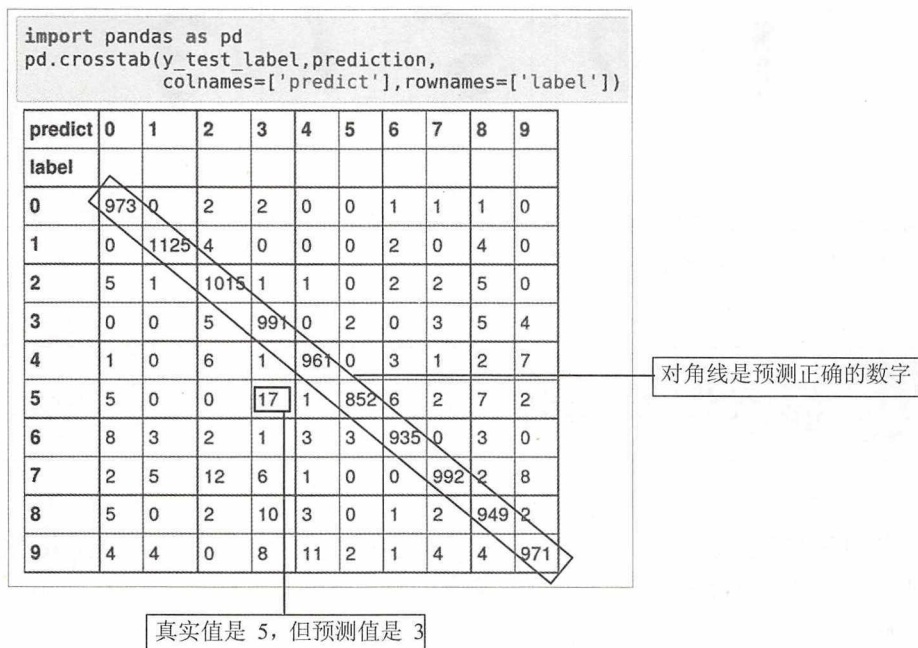


图 7-12

从以上混淆矩阵中，我们观察的结果如下：

- **对角线是预测正确的数字，我们发现：**真实值是“1”，被正确预测为“1”的项数有 1125 项，预测准确率最高，最不容易混淆。真实值是“5”，被正确预测为“5”的项数有 852 项最低，也就是说最容易混淆。
- **其他非对角线的数字代表将某一个标签预测错误，成为另一个标签，我们发现：**真实值是“5”，但是预测值是“3”。

2. 建立真实值与预测 DataFrame

因为我们希望能找出真实值是“5”但预测值是“3”的数据，所以创建下列 DataFrame。下面的程序代码用来创建 DataFrame，包含 label（真实值）与 prediction（预测值）。



```
In [22]: df = pd.DataFrame({'label':y_test_label, 'predict':prediction})
         df[:2]
```

```
Out[22]:
```

	label	predict
0	7	7
1	2	2

以上执行的结果有两个字段，分别是 label 与 predict。

3. 查询真实值是“5”但预测值是“3”的数据

Pandas DataFrame 可以很方便地让我们查询数据。例如下面的程序代码，可以找出真实值是“5”但预测值是“3”的数据。

```
In [23]: df[(df.label==5)&(df.predict==3)]
```

```
Out[23]:
```

	label	predict
340	5	3
1003	5	3
1393	5	3
2035	5	3
2526	5	3
2597	5	3
2810	5	3

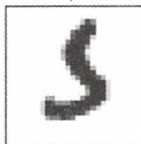
从以上执行结果可知共有 17 项，显示前 7 项结果。

4. 查看第 340 项数据

我们可以查看第 340 项结果，真实值是 5 但预测值为 3。

```
In [25]: plot_images_labels_prediction(x_test_image,y_test_label
                                         ,prediction,idx=340,num=1)
```

label=5,predict=3



从执行结果来看，这个数字图形看起来像 5 又像 3，所以预测错误。

7.8 隐藏层增加为 1000 个神经元

为了增加多层感知器模型的准确率，在本节的范例中将隐藏层原本 256 个神经元改为

1000。有关本节的完整程序代码，请参考范例程序 Keras_Mnist_MLP_h1_1000.ipynb。
我们将使用下列程序代码建立多层感知器模型，如图 7-13 所示。

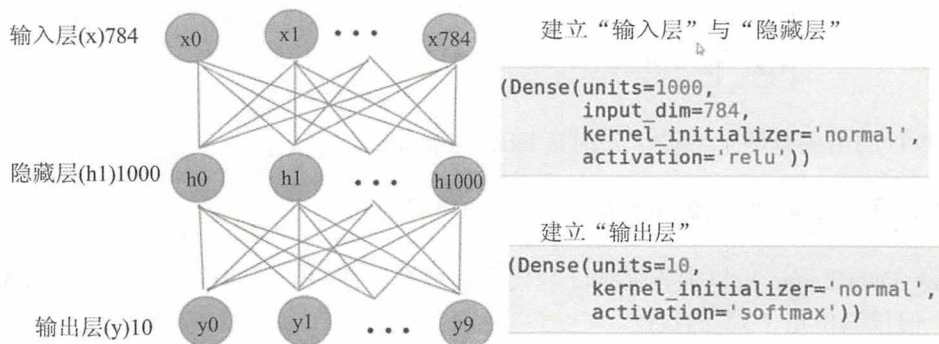


图 7-13

步骤01 将隐藏层原本 256 个神经元改为 1000 个神经元。

```
In [7]: model = Sequential()

In [8]: #将“输入层”与“隐藏层”加入模型

In [9]: model.add(Dense(units=1000,
                        input_dim=784,
                        kernel_initializer='normal',
                        activation='relu'))

In [10]: #将“输出层”加入模型

In [11]: model.add(Dense(units=10,
                        kernel_initializer='normal',
                        activation='softmax'))
```

原本是256, 改为1000

步骤02 查看模型的摘要。

我们可以使用下列指令来查看模型的摘要。

```
In [12]: print(model.summary())
```

执行后屏幕显示界面如图 7-14 所示。

Layer (type)	Output Shape	Param #	Connected to
dense_1 (Dense)	(None, 1000)	785000	dense_input_1[0][0]
dense_2 (Dense)	(None, 10)	10010	dense_1[0][0]
Total params: 795,010			
Trainable params: 795,010			
Non-trainable params: 0			

隐藏层共 1000 个神经元

输出层共 10 个神经元

图 7-14



步骤03 开始训练。

从指令执行后的屏幕显示界面中可以看到共执行了 10 个训练周期，如图 7-15 所示，从中可以发现误差越来越小，准确率越来越高。

```
In [14]: train_history=model.fit(x=x_Train_normalize,
                                y=y_Train_OneHot,validation_split=0.2,
                                epochs=10, batch_size=200,verbose=2)

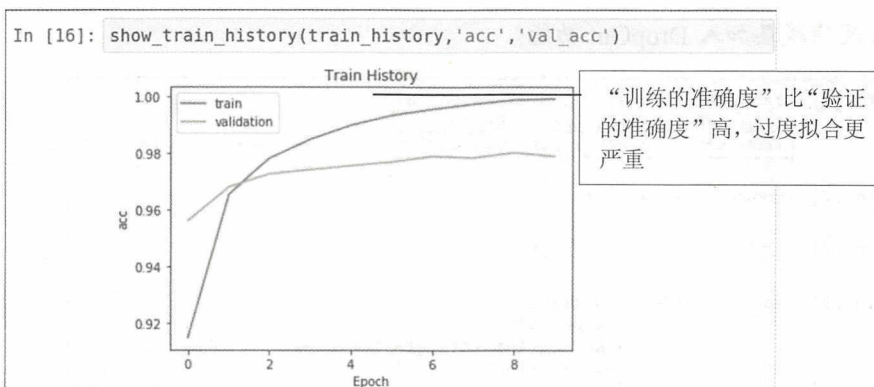
13s - loss: 0.2974 - acc: 0.9155 - val_loss: 0.1532 - val_acc: 0.9581
Epoch 2/10
15s - loss: 0.1195 - acc: 0.9649 - val_loss: 0.1089 - val_acc: 0.9680
Epoch 3/10
14s - loss: 0.0775 - acc: 0.9777 - val_loss: 0.0925 - val_acc: 0.9728
Epoch 4/10
14s - loss: 0.0533 - acc: 0.9846 - val_loss: 0.0875 - val_acc: 0.9743
Epoch 5/10
14s - loss: 0.0372 - acc: 0.9899 - val_loss: 0.0770 - val_acc: 0.9758
Epoch 6/10
13s - loss: 0.0266 - acc: 0.9932 - val_loss: 0.0761 - val_acc: 0.9771
Epoch 7/10
14s - loss: 0.0188 - acc: 0.9955 - val_loss: 0.0764 - val_acc: 0.9778
Epoch 8/10
14s - loss: 0.0139 - acc: 0.9966 - val_loss: 0.0713 - val_acc: 0.9795
Epoch 9/10
13s - loss: 0.0090 - acc: 0.9988 - val_loss: 0.0715 - val_acc: 0.9805
Epoch 10/10
14s - loss: 0.0065 - acc: 0.9991 - val_loss: 0.0740 - val_acc: 0.9795
```

图 7-15

步骤04 查看训练过程的准确率。

在执行结果界面，“acc 训练的准确率”是深色的，“val_acc 验证的准确率”是浅色的，总共执行了 10 个训练周期，我们可以发现：

- 无论是训练还是验证，准确率都越来越高。
- 在 Epoch 训练后期，“acc 训练的准确率”比“val_acc 验证的准确率”高，过度拟合更严重。



步骤05 预测准确率。

```
In [18]: scores = model.evaluate(x_Test_normalize, y_Test_OneHot)
          print()
          print('accuracy=',scores[1])

9728/10000 [=====>.] - ETA: 0s
accuracy= 0.9794
```

由以上执行结果可知准确率是 0.9794。

7.9 多层感知器加入 DropOut 功能以避免过度拟合

为了解决过度拟合的问题，在本节的范例中会加入 Dropout 功能。有关本节的完整程序代码，请参考范例程序 Keras_Mnist_MLP_h1_1000_}DropOut.ipynb。

下面建立多层感知器模型。

步骤01 隐藏层改为 1000 个神经元并且加入 DropOut 功能，如图 7-16 所示。

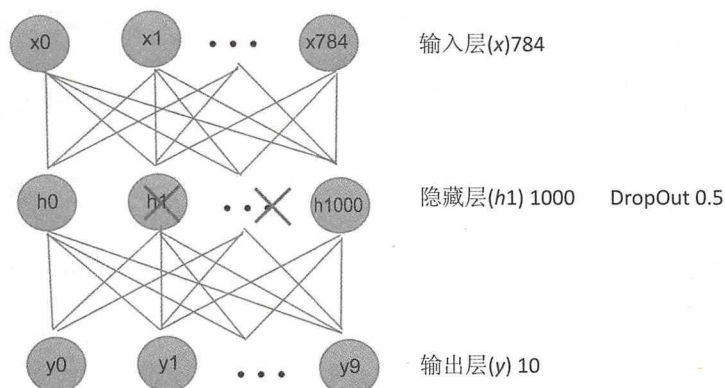


图 7-16

步骤02 修改隐藏层加入 DropOut 功能。

```
In [6]: from keras.models import Sequential
        from keras.layers import Dense
        from keras.layers import Dropout
```

导入 Dropout 模块

```
In [7]: model = Sequential()
```

```
In [8]: #将“输入层”与“隐藏层”加入模型
```

```
In [9]: model.add(Dense(units=1000,
                        input_dim=784,
                        kernel_initializer='normal',
                        activation='relu'))
```

```
In [10]: model.add(Dropout(0.5))
```

加入 Dropout 功能

```
In [11]: #将“输出层”加入模型
```

```
In [12]: model.add(Dense(units=10,
                        kernel_initializer='normal',
                        activation='softmax'))
```

步骤03 查看模型的摘要。

我们可以使用下列指令来查看模型的摘要。

```
In [13]: print(model.summary())
```

执行后界面如图 7-17 所示。

Layer (type)	Output Shape	Param #	Connected to
dense_1 (Dense)	(None, 1000)	785000	dense_input_1[0][0]
dropout_1 (Dropout)	(None, 1000)	0	dense_1[0][0]
dense_2 (Dense)	(None, 10)	10010	dropout_1[0][0]
Total params: 795,010			
Trainable params: 795,010			
Non-trainable params: 0			

加入DropOut 功能

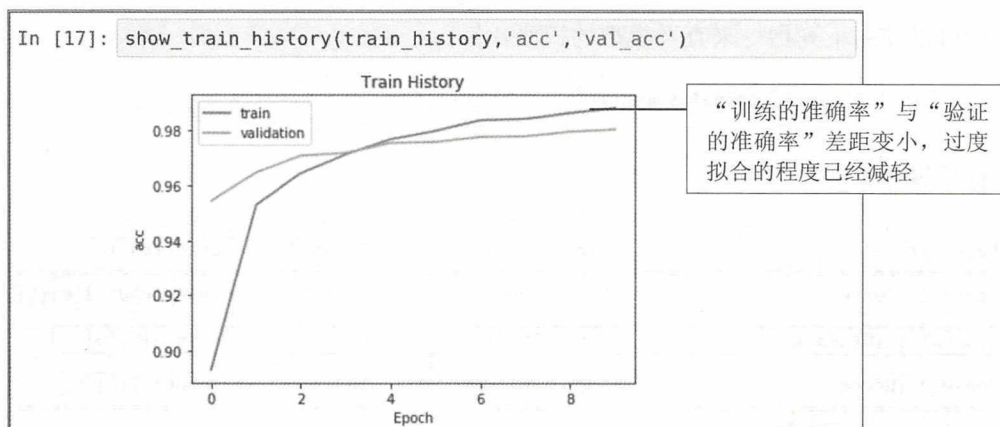
图 7-17

步骤04 查看训练过程的准确率。

```
In [15]: train_history=model.fit(x=x_Train_normalize,
                                y=y_Train_OneHot,validation_split=0.2,
                                epochs=10, batch_size=200,verbose=2)
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/10
13s - loss: 0.3573 - acc: 0.8934 - val_loss: 0.1620 - val_acc: 0.9544
Epoch 2/10
13s - loss: 0.1610 - acc: 0.9530 - val_loss: 0.1184 - val_acc: 0.9648
Epoch 3/10
16s - loss: 0.1173 - acc: 0.9645 - val_loss: 0.1001 - val_acc: 0.9709
Epoch 4/10
15s - loss: 0.0940 - acc: 0.9714 - val_loss: 0.0924 - val_acc: 0.9719
Epoch 5/10
17s - loss: 0.0766 - acc: 0.9770 - val_loss: 0.0817 - val_acc: 0.9756
Epoch 6/10
17s - loss: 0.0629 - acc: 0.9799 - val_loss: 0.0775 - val_acc: 0.9758
Epoch 7/10
14s - loss: 0.0543 - acc: 0.9839 - val_loss: 0.0788 - val_acc: 0.9778
Epoch 8/10
14s - loss: 0.0502 - acc: 0.9844 - val_loss: 0.0745 - val_acc: 0.9780
Epoch 9/10
14s - loss: 0.0436 - acc: 0.9865 - val_loss: 0.0692 - val_acc: 0.9798
Epoch 10/10
14s - loss: 0.0389 - acc: 0.9883 - val_loss: 0.0681 - val_acc: 0.9806
```

“acc 训练的准确率”与“val_acc 验证的准确率”差距很小，代表已经改善了过度拟合的问题

步骤05 图示训练过程的准确率。

在执行界面中，“acc 训练的准确率”是深色的，“val_acc 验证的准确率”是浅色的，总共执行了 10 个训练周期，我们可以发现：

- 无论是训练还是验证，准确率都越来越高。
- 在 Epoch 训练后期，虽然“acc 训练的准确率”比“val_acc 验证的准确率”高，但是“训练的准确率”与“验证的准确率”差距变小，过度拟合的程度已经减轻。

步骤06 查看准确率。

```
In [19]: scores = model.evaluate(x_Test_normalize, y_Test_OneHot)
          print()
          print('accuracy=', scores[1])

          9824/10000 [=====>.] - ETA: 0s
          accuracy= 0.9802
```

从以上执行结果可知准确率是 0.98。比之前未加入 Dropout 时还高，这代表加入了 Dropout 不但可以解决过度拟合的问题，还可以增加准确率。

7.10 建立多层感知器模型包含两个隐藏层

为了更进一步增加多层感知器模型的准确率，在本节的范例中将建立两个隐藏层。有关完整的程序码，请参考范例程序 `Keras_Mnist_MLP_h1000_DropOut_h1000_DropOut.ipynb`。

步骤01 加入两个隐藏层并且加入 Dropout 功能。

如图 7-18 所示，我们将加入两个隐藏层并且加入 Dropout 功能。

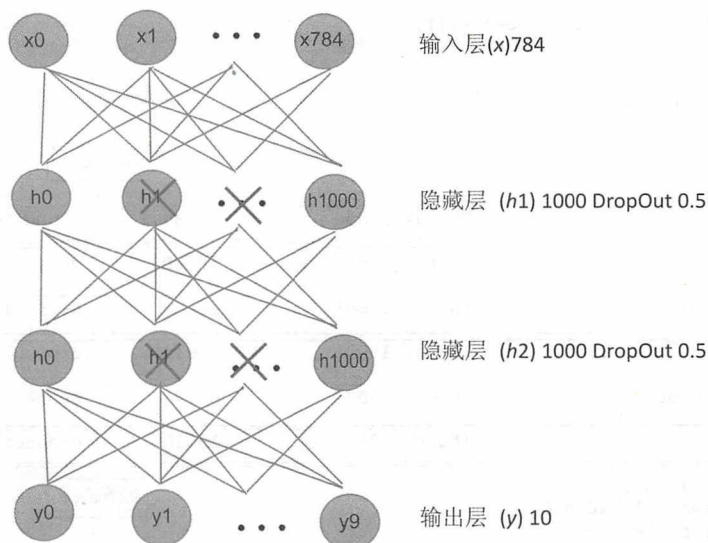


图 7-18

➤ 建立模型

```
In [7]: model = Sequential()
```

➤ 加入“输入层”与“隐藏层 1”

```
In [9]: model.add(Dense(units=1000,
                        input_dim=784,
                        kernel_initializer='normal',
                        activation='relu'))
```

```
In [10]: model.add(Dropout(0.5))
```

➤ 加入“隐藏层 2”

```
In [12]: model.add(Dense(units=1000,
                        kernel_initializer='normal',
                        activation='relu'))
```

```
In [13]: model.add(Dropout(0.5))
```

➤ 加入“输出层”

```
In [15]: model.add(Dense(units=10,
                        kernel_initializer='normal',
                        activation='softmax'))
```

步骤02 查看模型的摘要。

我们可以使用下列指令来查看模型的摘要。

```
In [13]: print(model.summary())
```

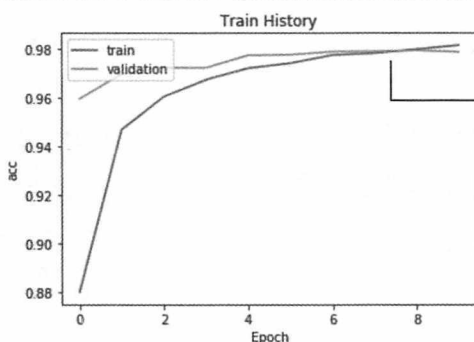
执行后屏幕显示界面如图 7-19 所示。

Layer (type)	Output Shape	Param #	Connected to
dense_1 (Dense)	(None, 1000)	785000	dense_input_1[0][0]
dropout_1 (Dropout)	(None, 1000)	0	dense_1[0][0]
dense_2 (Dense)	(None, 1000)	1001000	dropout_1[0][0]
dropout_2 (Dropout)	(None, 1000)	0	dense_2[0][0]
dense_3 (Dense)	(None, 10)	10010	dropout_2[0][0]
Total params: 1,796,010			
Trainable params: 1,796,010			
Non-trainable params: 0			

图 7-19

步骤03 查看训练过程的准确率。

```
In [20]: show_train_history(train_history, 'acc', 'val_acc')
```



“训练的准确率”与“验证的准确率”差距变小，代表已经大致解决了过度拟合的问题

在执行界面中，“acc 训练的准确率”是深色的，“val_acc 验证的准确率”是浅色的，总共执行了 10 个训练周期，我们可以发现：

- 无论是训练还是验证，准确率都越来越高。
- 在 Epoch 训练后期，虽然“acc 训练的准确率”比“val_acc 验证的准确率”高，但是“训练的准确率”与“验证的准确率”差距变小，这代表已经大致解决了过度拟合的问题。

步骤04 查看准确率。

```
In [22]: scores = model.evaluate(x_Test_normalize, y_Test_OneHot)
          print()
          print('accuracy=', scores[1])

          9984/10000 [=====>.] - ETA: 0s
          accuracy= 0.9797
```




从以上执行结果可知准确率是 0.9797，准确率并没有显著提升。

7.11 结论

在本章中，我们使用多层感知器模型来识别 MNIST 数据集中的手写数字，尝试将模型加宽、加深，以提高准确率，并且加入 Drop 层，以避免过度拟合，准确率接近 0.98。不过，多层感知器有其极限，如果还要进一步提升准确率，就必须使用卷积神经网络。

第8章

Keras卷积神经网络 识别手写数字

在本章中，我们将介绍使用 Keras 建立卷积神经网络（Convolutional Neural Network, CNN），然后训练模型、评估模型准确率接近 0.99，最后使用训练完成的模型来识别 MNIST 手写数字。

卷积神经网络是由一位计算机科学家 Yann LeCun 所提出的，他在机器学习、计算机视觉和计算神经科学等诸多领域都有贡献。

有关本章完整的程序代码，请参考范例程序 Keras_Mnist_CNN.ipynb。范例程序下载与安装，请参考本书附录 A 中的“本书范例程序的下载与安装说明”。

8.1 卷积神经网络简介

1. 多层感知器与卷积神经网络

如图 8-1 所示，多层感知器与卷积神经网络主要的差异是：卷积神经网络增加了卷积层 1、池化层 1、卷积层 2、池化层 2 的处理来提取特征。

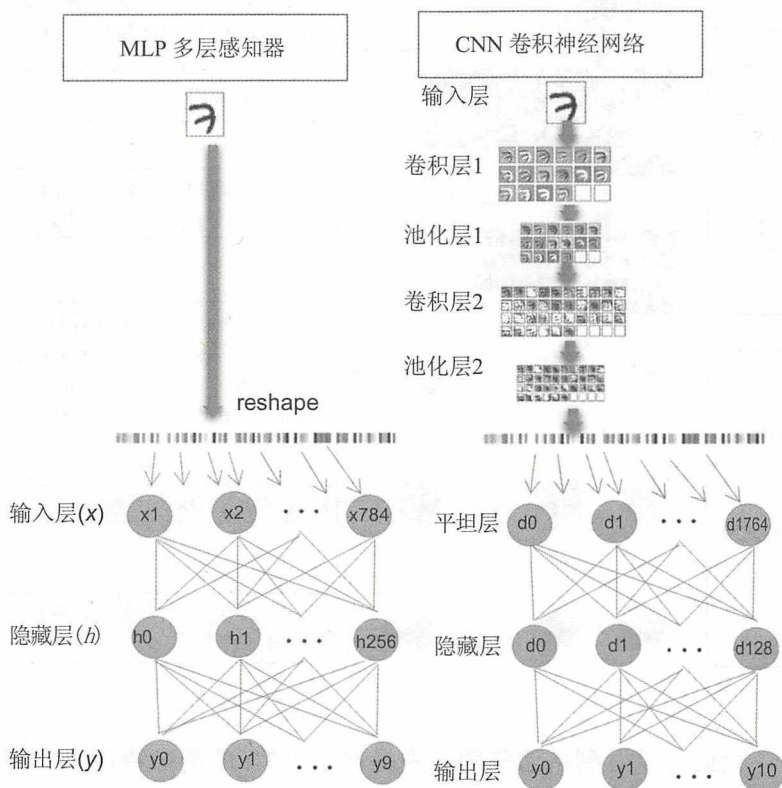


图 8-1

2. 卷积神经网络介绍

从图 8-1 中可以看到卷积神经网络可分为两大部分。

● 图像的特征提取

通过卷积层 1、池化层 1、卷积层 2、池化层 2 提取图像的特征。

● 完全连接的神经网络

包含平坦层、隐藏层、输出层，所组成的类神经网络，如图 8-2 所示。

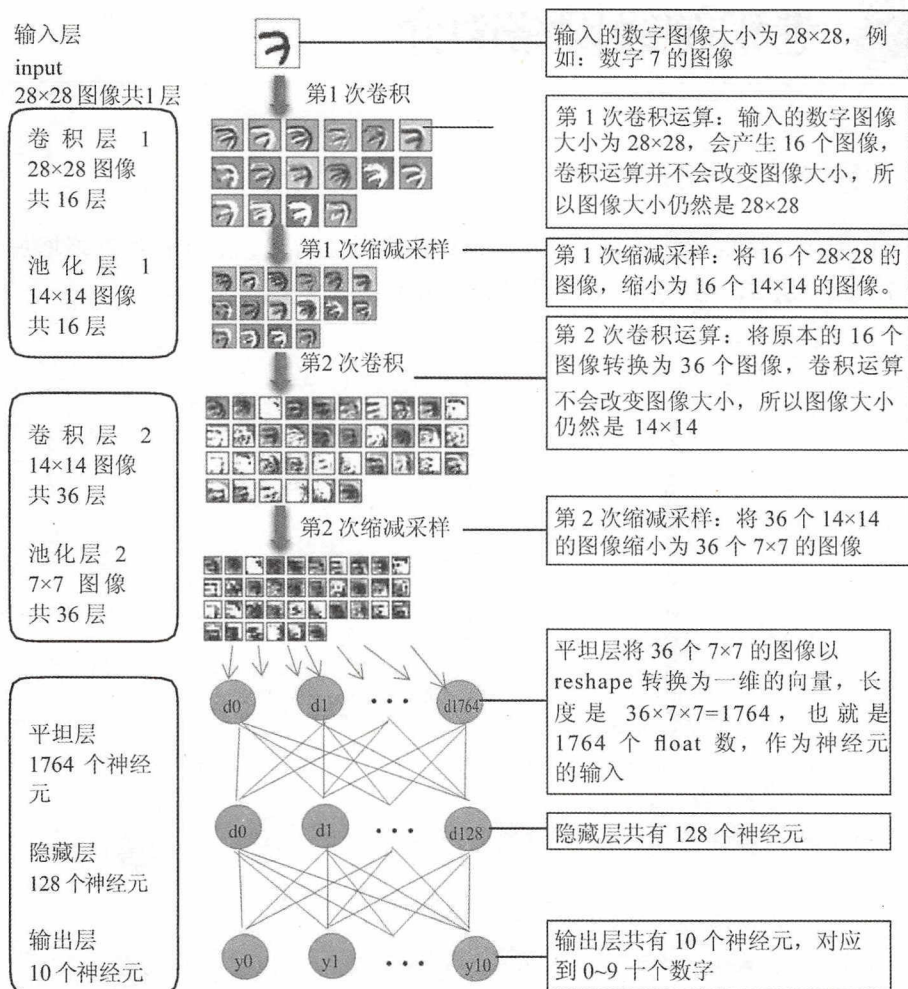


图 8-2

从图 8-2 中，我们可以看到从这些图像中提取了“7”的图像特征，卷积运算的效果类似滤镜效果，即用于提取不同的特征。注：downsampling 就是缩减像素采样，简称缩减采样。

3. 卷积运算

卷积层的意义是将原本一个图像经过卷积运算产生多个图像，就好像将相片叠加起来。

➤ 卷积运算的运算方式（见图 8-3）

- (1) 先以随机的方式产生，filter weight 大小是 3×3 。
- (2) 要转换的图像从左到右、自上而下，按序选取 3×3 的矩阵。
- (3) 图像选取的矩阵 (3×3) 与 filter weight (3×3) 乘积，计算产生第 1 行、第 1 列的数字。

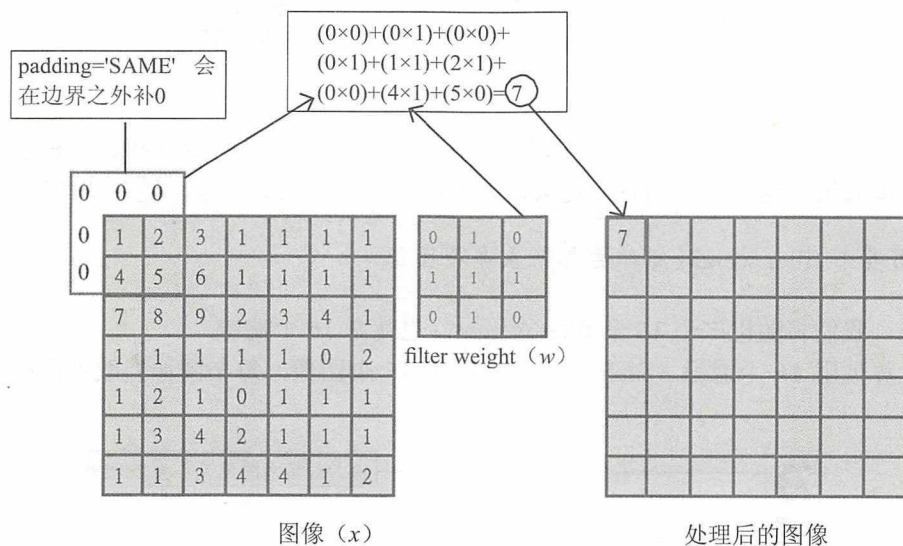


图 8-3

再以相同的方式计算第 1 行、第 2 列的数字，如图 8-4 所示。

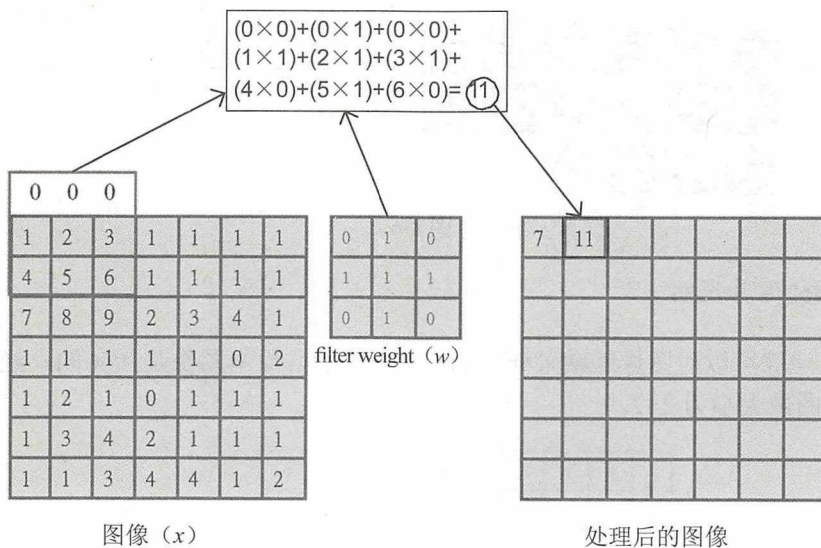


图 8-4

按照上面的相同方式，按序完成所有运算，就可以完成图像的处理。

4. 使用单个 filter weight 卷积运算产生图像

如图 8-5 所示，将大小为 28×28 的数字图像 7，使用随机产生的 5×5 filter weight (w) 滤镜进行卷积运算。

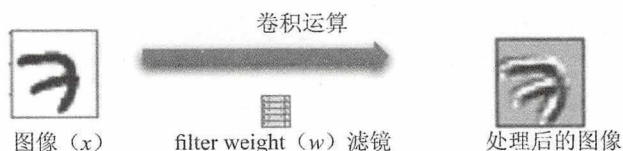


图 8-5

卷积运算并不会改变图像大小，所以处理后的图像大小仍然是 28×28 。卷积运算后的效果很类似滤镜效果，这可以帮助我们提取输入的不同特征，例如边缘、线条和角等。

5. 使用多个 filter weight 卷积运算产生多个图像

接下来，我们将随机产生 16 个 filter weight，也就是 16 个滤镜。

卷积运算使用 16 个滤镜 (filter weight) 产生 16 个图像，每个图像提取不同的特征，如图 8-6 所示。

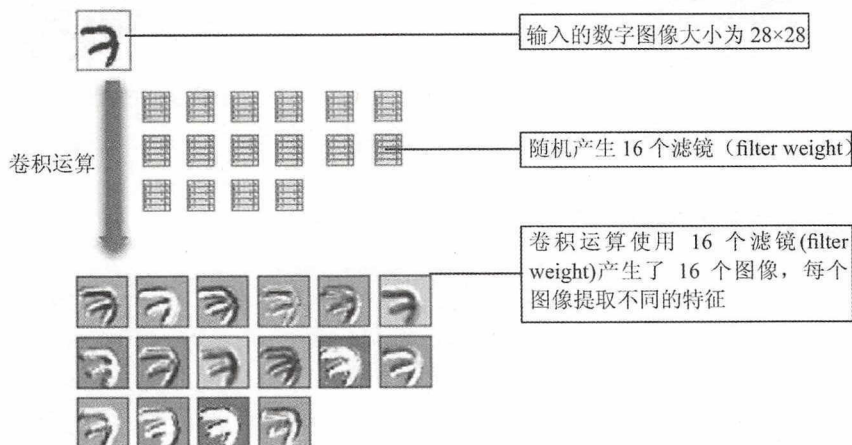


图 8-6

6. Max-Pool 运算说明

Max-Pool 运算可以对图像缩减采样，如图 8-7 所示，原本图像是 4×4 的，经过 Max-Pool 运算转换后，图像大小为 2×2 。

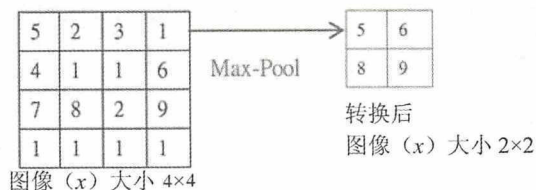


图 8-7

以上 Max-Pool 运算详细说明如下。

- 在左上角 4 个数字：5、2、4、1 最大的是 5，所以计算结果是 5，如图 8-8 所示。

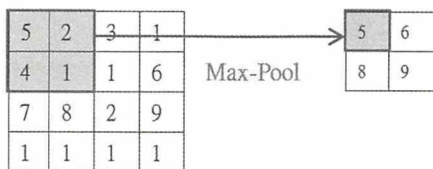


图 8-8

- 在右上角 4 个数字：3、1、1、6，最大的是 6，所以计算结果是 6，如图 8-9 所示。

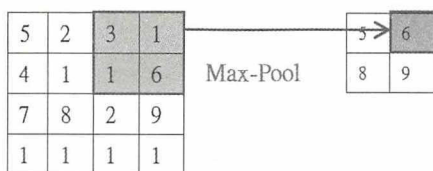


图 8-9

- 在左下角 4 个数字：7、8、1、1，最大的是 8，所以计算结果是 8，如图 8-10 所示。

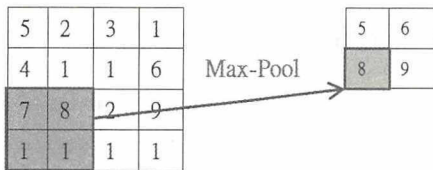


图 8-10

- 在右下角 4 个数字：2、9、1、1，最大的是 9，所以计算结果是 9，如图 8-11 所示。

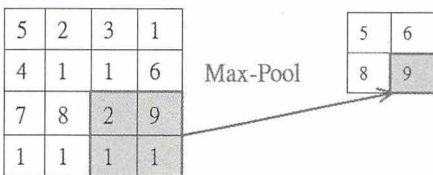


图 8-11

7. 使用 Max-Pool 转换手写数字图像

使用 Max-Pool 缩减采样，进行手写数字图像转换，将 16 个 28×28 的图像缩小为 16 个 14×14 的图像，但是不会改变图像的数量（仍然是 16 个），如图 8-12 所示。

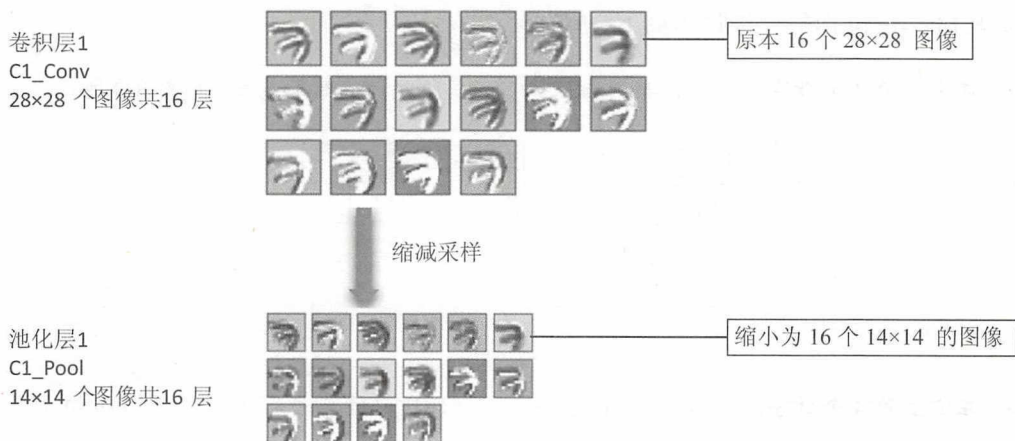


图 8-12

缩减采样会缩小图像，有下列好处。

- (1) **减少需处理的数据点**：减少后续运算所需的时间。
- (2) **让图像位置差异变小**：例如手写数字 7，位置上下左右可能不同，位置的不同可能会影响识别。减小图像大小，让数字的位置差异变小。
- (3) **参数的数量和计算量下降**：这在一定程度上也控制了过度拟合。

8. 建立卷积神经网络识别 MNIST 数据集

建立卷积神经网络识别 MNIST 数据集的步骤如图 8-13 所示。



图 8-13

8.2 进行数据预处理

卷积神经网络与多层感知器进行数据预处理的方式不同，说明见表 8-1。

表 8-1 卷积神经网络与多层感知器进行数据预处理对比

	reshape	说明
多层感知器	<code>image.reshape(60000,784)</code>	多层感知器因为直接送进神经元处理，所以 reshape 转换为 60 000 项，每一项有 784 个数字，作为 784 个神经元的输入
卷积神经网络	<code>image.reshape(60000,28,28,1)</code>	卷积神经网络因为必须先进行卷积与池化运算，所以必须保持图像的维数，所以 reshape 转换为 60 000 项，每一项是 28×28×1 的图像，分别是 28（宽）×28（高）×1（单色）

步骤01 导入所需模块。

```
In [1]: from keras.datasets import mnist
        from keras.utils import np_utils
        import numpy as np
        np.random.seed(10)

        Using TensorFlow backend.
```

步骤02 读取 MNIST 数据。

```
In [2]: (x_Train, y_Train), (x_Test, y_Test) = mnist.load_data()
```

步骤03 将 features（数字图像特征值）转换为四维矩阵。

将 features（数字图像特征值）以 reshape 转换为 6000×28×28×1 的 4 维矩阵。

```
In [3]: x_Train4D=x_Train.reshape(x_Train.shape[0],28,28,1).astype('float32')
        x_Test4D=x_Test.reshape(x_Test.shape[0],28,28,1).astype('float32')
```

步骤04 将 features（数字图像特征值）标准化。

将 features（数字图像特征值）标准化可以提高模型预测的准确度，并且更快收敛。

```
In [4]: x_Train4D_normalize = x_Train4D / 255
        x_Test4D_normalize = x_Test4D / 255
```

步骤05 label（数字真实的值）以 One-Hot Encoding 进行转换。

使用 `np_utils.to_categorical` 将训练数据与测试数据的 label 进行 One-Hot Encoding（一位有效编码）转换。


```
In [5]: y_TrainOneHot = np_utils.to_categorical(y_Train)
        y_TestOneHot = np_utils.to_categorical(y_Test)
```

8.3 建立模型

我们将使用下列程序代码来建立卷积神经网络，如图 8-14 所示。

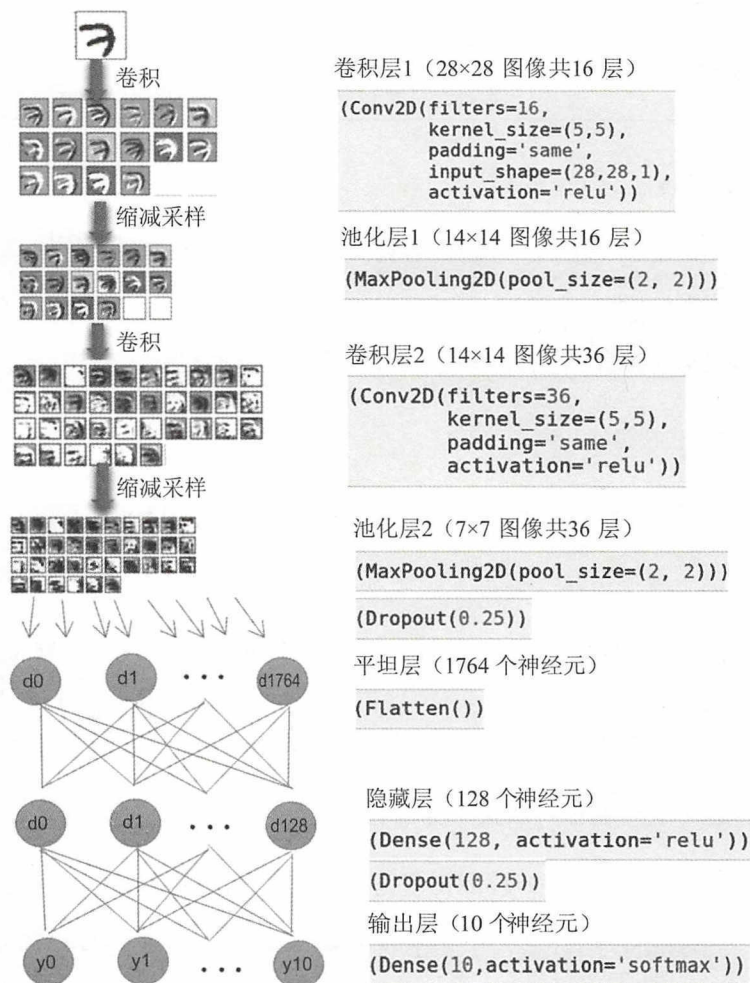


图 8-14

1. 导入所需模块

```
In [6]: from keras.models import Sequential
        from keras.layers import Dense,Dropout,Flatten,Conv2D,MaxPooling2D
```



程序代码说明见表 8-2。

表 8-2 程序代码说明

程序代码	说明
<code>from keras.models import Sequential</code>	导入 MNIST 数据集
<code>From keras.layers import Dense,Dropout, Flatten,Conv2D,MaxPooling2D</code>	导入 keras 的 layers 模块，后续卷积神经网络会使用

2. 建立 keras 的 Sequential 模型

建立一个 Sequential 线性堆叠模型，后续只需要使用 `model.add()` 方法将各个神经网络层加入模型即可。

```
In [7]: model = Sequential()
```

3. 建立卷积层 1 与池化层 1

一个完整的卷积运算包含一个卷积层与一个池化层。

➤ 建立卷积层 1

使用下列程序代码建立卷积层 1。输入的数字图像大小为 28×28 ，进行第 1 次卷积运算会产生 16 个图像，卷积运算并不会改变图像大小，所以图像大小仍然是 28×28 。

```
In [8]: model.add(Conv2D(filters=16,
                        kernel_size=(5,5),
                        padding='same',
                        input_shape=(28,28,1),
                        activation='relu'))
```

以上程序代码把 Conv2D 层加入模型中，需输入表 8-3 中的参数。

表 8-3 把 Conv2D 层加入模型中需输入的参数

程序代码	说明
<code>filters=16,</code>	建立 16 个滤镜
<code>kernel_size=(5,5),</code>	每一个滤镜 5×5 大小
<code>padding='same',</code>	此设置让卷积运算产生的卷积图像大小不变
<code>input_shape=(28, 28,1),</code>	第一、二维：代表输入的图像形状为 28×28 第三维：因为是单色灰度图像，所以最后维数值是 1
<code>activation='relu'</code>	设置 ReLU 激活函数

➤ 建立池化层 1

下面的程序代码建立池化层 1，输入参数 `pool_size=(2,2)`，执行第 1 次缩减采样，将 16 个 28×28 的图像缩小为 16 个 14×14 的图像。

```
In [9]: model.add(MaxPooling2D(pool_size=(2, 2)))
```

4. 建立卷积层 2 与池化层 2

➤ 建立卷积层 2

使用下面的程序代码建立卷积层 2。执行第 2 次卷积运算：将原本的 16 个图像转换为 36 个图像，卷积运算不会改变图像大小，所以图像大小仍然是 14×14。

```
In [10]: model.add(Conv2D(filters=36,
                           kernel_size=(5,5),
                           padding='same',
                           activation='relu'))
```

以上程序代码把 Conv2D 层加入模型中，需输入表 8-4 中的参数。

表 8-4 把 Conv2D 层加入模型中需输入的参数

程序代码	说明
filters=36,	建立 36 个滤镜
kernel_size=(5,5),	每一个滤镜的大小为 5×5
padding='same',	此设置让卷积运算并不会改变图像的大小
activation='relu'	设置 ReLU 激活函数

➤ 建立池化层 2，并且加入 Dropout 避免过度拟合

下面的程序代码建立池化层 2，输入参数 pool_size=(2,2)，执行第 2 次缩减采样，将 36 个 14×14 的图像缩小为 36 个 7×7 的图像。

```
In [11]: model.add(MaxPooling2D(pool_size=(2, 2)))
```

下面的程序代码把 Dropout(0.25)层加入模型中。其功能是，每次训练迭代时，会随机在神经网络中放弃 25% 的神经元，以避免过度拟合。

```
In [12]: model.add(Dropout(0.25))
```

5. 建立神经网络（平坦层、隐藏层、输出层）

➤ 建立平坦层

以下程序代码建立平坦层，将之前的步骤已经建立的池化层 2，共有 36 个 7×7 的图像转换为一维的向量，长度是 36×7×7=1764，也就是 1764 个 Float 数，正好对应 1764 个神经元。

```
In [13]: model.add(Flatten())
```

➤ 建立隐藏层

下面的程序代码建立隐藏层，共有 128 个神经元。

```
In [14]: model.add(Dense(128, activation='relu'))
```


并且把 Dropout 层加入模型中。Dropout(0.5)的功能是，每次训练迭代时，会随机地在神经网络中放弃 50% 的神经元，以避免过度拟合。

```
In [15]: model.add(Dropout(0.5))
```

➤ 建立输出层

最后建立输出层，共有 10 个神经元，对应 0~9 共 10 个数字。并且使用 softmax 激活函数进行转换，softmax 可以将神经元的输出转换为预测每一个数字的概率。

```
In [16]: model.add(Dense(10,activation='softmax'))
```

6. 查看模型的摘要

我们可以使用下列指令来查看模型的摘要，如图 8-15 所示。

```
In [17]: print(model.summary())
```

Layer (type)	Output Shape	Param #	
conv2d_1 (Conv2D)	(None, 28, 28, 16)	416	卷积层1 与池化层1
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 16)	0	
conv2d_2 (Conv2D)	(None, 14, 14, 36)	14436	卷积层2 与池化层2
max_pooling2d_2 (MaxPooling2D)	(None, 7, 7, 36)	0	
dropout_1 (Dropout)	(None, 7, 7, 36)	0	
flatten_1 (Flatten)	(None, 1764)	0	神经网络（平坦层、隐藏层、输出层）
dense_1 (Dense)	(None, 128)	225920	
dropout_2 (Dropout)	(None, 128)	0	
dense_2 (Dense)	(None, 10)	1290	
Total params: 242,062.0			
Trainable params: 242,062.0			
Non-trainable params: 0.0			

图 8-15

8.4 进行训练

当我们建立好深度学习模型后，就可以使用反向传播算法（参考第 2 章）进行训练。

1. 定义训练方式

在训练模型之前，我们必须使用 compile 方法对训练模型进行设置，如下列指令：

```
In [18]: model.compile(loss='categorical_crossentropy',  
                      optimizer='adam',metrics=['accuracy'])
```

compile 方法需输入 3 个参数: loss、optimizer 和 metrics, 对这 3 个参数的解释说明可参考 7.4 节。

2. 开始训练

执行训练的程序代码如下:

```
In [19]: train_history=model.fit(x=x_Train4D_normalize,  
                                y=y_TrainOneHot,validation_split=0.2,  
                                epochs=10, batch_size=300,verbose=2)
```

使用 model.fit 进行训练, 训练过程会存储在 train_history 变量中, 这个训练需输入下列参数。

(1) 输入训练数据参数

- `x=x_Train4D_normalize` (features 数字图像的特征值)。
- `y=y_Train_OneHot` (label 数字图像真实的值)。

(2) 设置训练与验证数据比例

- 设置参数 `validation_split=0.2`。

训练之前 Keras 会自动将数据分成: 80%作为训练数据, 20%作为验证数据。因为全部是 60 000 项, 所以分成: $60\,000 \times 0.8 = 48\,000$ 作为训练数据, $60\,000 \times 0.2 = 12\,000$ 作为验证数据。

(3) 设置训练周期次数与每一批次项数

- `epochs=10`: 执行 10 个训练周期。
- `batch_size=300`: 每一批次 300 项数据。

(4) 设置显示训练过程

- `verbose=2`: 显示训练过程。

以上程序代码共执行了 10 个训练周期, 每一个训练周期执行下列功能:

- 使用 48 000 项训练数据进行训练, 分为每一批次 300 项, 所以大约分为 160 批次 ($48\,000/300=160$) 进行训练。
- Epoch (训练周期) 训练完成后, 会计算这个训练周期的准确率与误差, 并且在 train_history 中新增一项数据记录。

以上程序代码执行后的结果如图 8-16 所示。

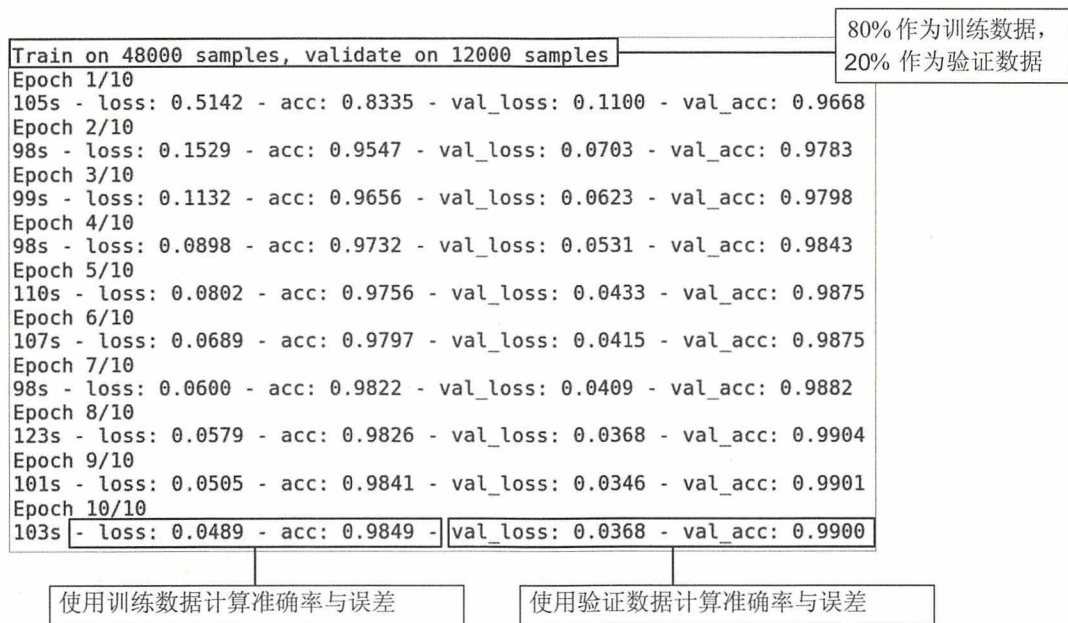


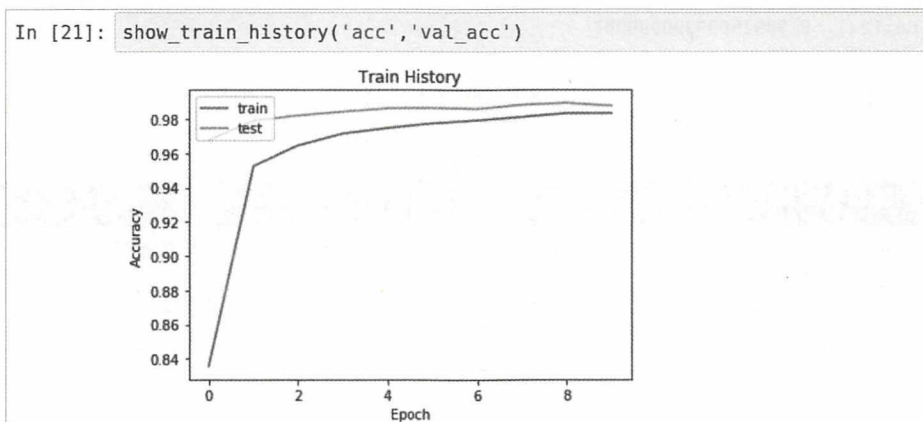
图 8-16

从以上执行结果的屏幕显示界面中可以看到共执行了 10 个训练周期，从中可以发现误差越来越小，准确率越来越高。

3. 画出准确率执行结果

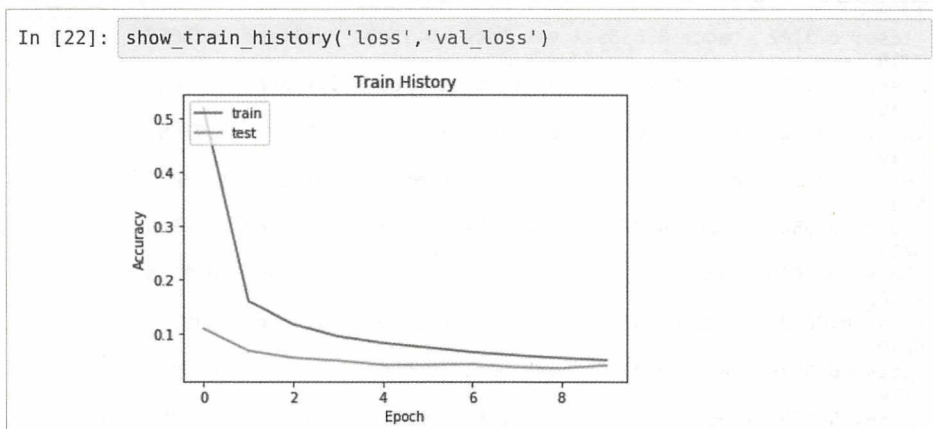
之前的训练步骤会将每一个训练周期的准确率与误差记录在 `train_history` 变量中。

我们可以使用下面的程序代码读取 `train_history`，画出准确率的执行结果。有关 `show_train_history` 的细节可参考第 7 章的说明。



在以上执行界面中，“acc 训练的准确率”是深色的，“val_acc 验证的准确率”是浅色的，总共执行了 10 个训练周期，我们可以发现，无论是训练还是验证，准确率都越来越高。

4. 画出误差执行结果



在以上执行界面中，“loss 训练的误差”是深色的，“val_loss 验证的误差”是浅色的，总共执行了 10 个训练周期，我们可以发现，无论是训练还是验证，验证的误差都越来越低。

8.5 评估模型准确率

在之前的步骤中，我们已经完成了训练，现在要使用 test 测试数据集来评估模型准确率。用下面的程序代码来评估模型的准确率。

```
In [23]: scores = model.evaluate(x_Test4D_normalize, y_TestOneHot)
          scores[1]

Out[23]: 0.9891999999999997
```

从以上的执行结果可知准确率是 0.989。程序代码说明见表 8-5。

表 8-5 程序代码说明

程序代码	说明
<code>scores = model.evaluate(</code>	使用 <code>model.evaluate</code> 评估模型的准确率，评估后的准确率会存储在 <code>scores</code> 中
<code>x= x_Test4D_normalize,</code>	测试数据的 features（已标准化测试数据的数字图像）
<code>y= y_Test_OneHot)</code>	测试数据的 label（数字图像真实的值）

8.6 进行预测

之前的步骤我们建立了模型，并且完成了训练模型，准确率达到 0.989，接下来将使用此

模型进行预测。

步骤01 执行预测。

我们可以使用下列指令执行预测。

```
In [24]: prediction=model.predict_classes(x_Test4D_normalize)
          9984/10000 [=====>.] - ETA: 0s
```

下面的程序代码使用 `model.predict_classes` 输入参数 `x_Test4D_normalize`（已标准化测试数据的数字图像）进行预测。

步骤02 预测结果。

我们可以使用下列指令查看预测结果的前 10 项数据。

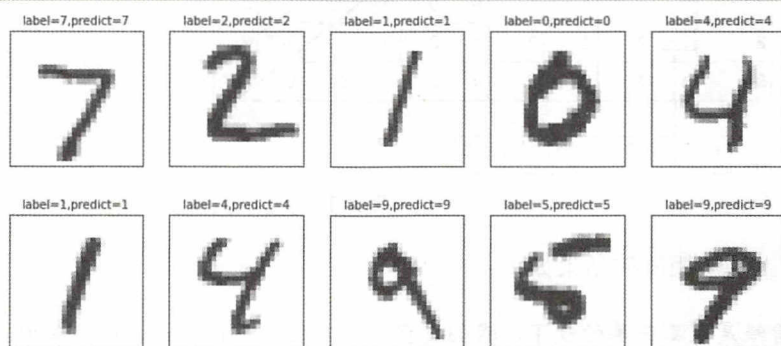
```
In [25]: prediction[:10]
Out[25]: array([7, 2, 1, 0, 4, 1, 4, 9, 5, 9])
```

可以看到第 1 项预测结果是 7，第 2 项是 2……

步骤03 显示前 10 项预测结果。

使用第 6 章创建的 `show_images_labels_prediction` 函数显示前 10 项预测结果，传入测试数据图像、label（真实值）及 predict（预测结果）。

```
In [27]: plot_images_labels_prediction(x_Test,y_Test,prediction,idx=0)
```



8.7 显示混淆矩阵

Pandas 提供了建立混淆矩阵的功能。

```
In [28]: import pandas as pd
pd.crosstab(y_Test, prediction,
            rownames=['label'], colnames=['predict'])
```

以上程序代码说明见表 8-6。

表 8-6 程序代码说明

程序代码	说明
<code>import pandas as pd</code>	导入 pandas 模块，后续会以 pd 来引用
<code>pd.crosstab(y_test_label, prediction, rownames=['label'], colnames=['predict'])</code>	使用 pd.crosstab 建立混淆矩阵，输入下列参数：测试数字图像的真实值 测试数字图像的预测结果 设置行的名称是 label 设置列的名称是 predict

执行后显示如图 8-17 所示的混淆矩阵。



图 8-17

观察以上混淆矩阵的结果如下：

- 对角线是预测正确的数字，我们发现：真实值是“1”，被正确预测为“1”的项数有 1132 项，最高，即最不容易混淆。真实值是“5”，被正确预测为“5”的项数有 881 项，最低，也就是说最容易混淆。
- 其他非对角线的数字代表将某一个标签错误预测为另一个标签，我们发现：真实值是“8”，但是预测值是“0”时最高，也就是最容易混淆。



8.8

结论

在本章中，我们使用卷积神经网络来识别 MNIST 数据集中的手写数字，其分类精度接近 0.99。不过，这只是单色手写数字的识别，相对来说比较简单，下一章我们将介绍更具挑战性的，使用卷积神经网络来识别 CIFAR-10 数据集，识别彩色图像共 10 个分类：飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船、卡车。

第9章

Keras CIFAR-10图像识别数据集

CIFAR-10 是由 Alex Krizhevsky、Vinod Nair 与 Geoffrey Hinton 收集的一个用于图像识别的数据集，共有 10 个分类：飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船、卡车。CIFAR-10 数据集与之前的 MNIST 数据集相比，它的色彩、颜色噪点较多，同一分类（如卡车）大小不一、角度不同、颜色不同。所以 CIFAR-10 图像识别的难度比 MNIST 数据集高很多。

我们可以在下列网址查看 CIFAR-10 数据集（见图 9-1）：

<https://www.cs.toronto.edu/~kriz/cifar.html>

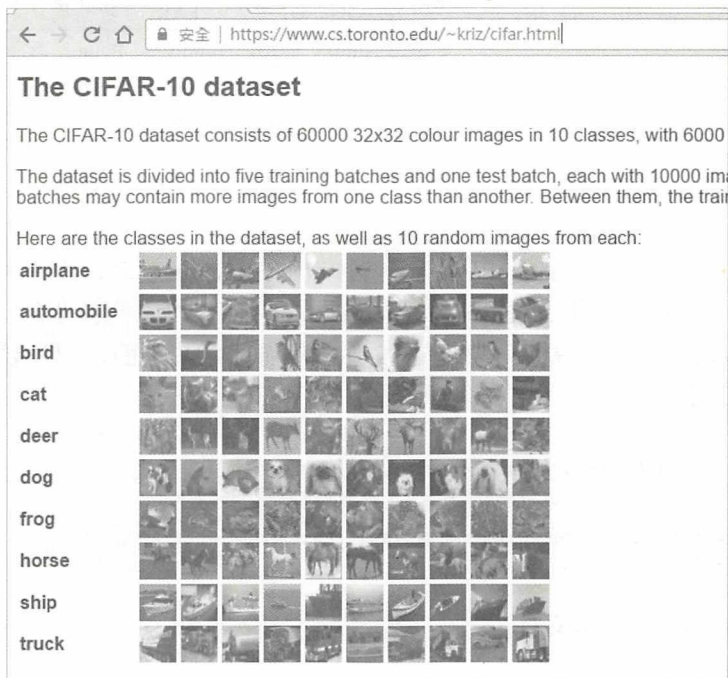


图 9-1

CIFAR-10 数据集共有 60 000 个 32×32 的彩色图像，有 50 000 个训练图像和 10 000 个测试图像。共有 10 个分类，它们是：飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船、卡车。每个分类有 6 000 个图像。有关完整的程序代码，参考范例程序 `Keras_Cifar_CNN_Introduce.ipynb`。有关范例程序下载与安装的细节，可参考本书附录 A 中的“本书范例程序的下载与安装说明”。

9.1 下载 CIFAR-10 数据

我们将创建 Keras 程序，下载并读取 CIFAR-10 数据。

步骤01 导入所需模块。

```
In [1]: from keras.datasets import cifar10
import numpy as np
np.random.seed(10)

Using TensorFlow backend.
```

程序代码说明见表 9-1。

表 9-1 程序代码说明

程序代码	说明
<code>from keras.datasets import cifar10</code>	从 <code>keras.datasets</code> 导入 CIFAR-10 数据集
<code>import numpy as np</code>	导入 NumPy 模块, NumPy 是 Python 语言的扩展程序库, 支持维度数组与矩阵运算
<code>np.random.seed(10)</code>	设置 seed 以生成需要的随机数

步骤02 下载并且解压缩 CIFAR-10 文件

Keras 提供了 `cifar10.load_data()` 用于下载或读取 CIFAR-10 数据。第一次执行 `cifar10.load_data()` 方法时, 程序会检查是否有 `cifar-10-batches-py.tar` 文件, 如果还没有, 就会下载文件, 并且解压缩下载的文件。以下是第一次下载文件时屏幕显示的界面, 因为要下载文件, 所以运行时间可能会比较长。

```
In [2]: (x_img_train,y_label_train), \
        (x_img_test, y_label_test)=cifar10.load_data()

Downloading data from http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Untaring file...
```

步骤03 查看 CIFAR-10 数据文件。

查看下载的 CIFAR-10 数据文件, 这个文件会因我们使用的环境是 Windows 或 Linux Ubuntu 而有所不同, 说明如下:

➤ 在 Windows 下查看下载文件的 CIFAR-10 数据文件

如图 9-2 所示, 因为笔者的用户名称是 kevin, 所以下载后会存放在目录 `C:\Users\kevin\.keras\datasets` 中, 文件名是 `cifar-10-batches-py.tar`。



图 9-2

➤ 在 Linux Ubuntu 下查看已下载的 MNIST 数据文件

下载完成后, 我们可以输入下列指令来查看 “`~/.keras/datasets/cifar-10-batches-py`” 下的 CIFAR-10 子目录, 如图 9-3 所示。

```
ll ~/.keras/datasets/cifar-10-batches-py
```

```

user@Ubuntu1604: ~
user@Ubuntu1604:~$ ll ~/.keras/datasets/cifar-10-batches-py
总计 181884
drwxr-xr-x 2 user user 4096 6月 5 2009 ./
drwxrwxr-x 3 user user 4096 3月 11 10:43 ../
-rw-r--r-- 1 user user 158 3月 31 2009 batches.meta
-rw-r--r-- 1 user user 31035704 3月 31 2009 data_batch_1
-rw-r--r-- 1 user user 31035320 3月 31 2009 data_batch_2
-rw-r--r-- 1 user user 31035999 3月 31 2009 data_batch_3
-rw-r--r-- 1 user user 31035696 3月 31 2009 data_batch_4
-rw-r--r-- 1 user user 31035623 3月 31 2009 data_batch_5
-rw-r--r-- 1 user user 88 6月 5 2009 readme.html
-rw-r--r-- 1 user user 31035526 3月 31 2009 test_batch
user@Ubuntu1604:~$

```

图 9-3

步骤04 读取 CIFAR-10 数据。

再次执行 `cifar10.load_data()` 时，由于之前已经下载了文件，不需要再下载了，只需要读取文件，因此运行时间不会太长。

```
In [2]: (x_img_train,y_label_train), \
        (x_img_test, y_label_test)=cifar10.load_data()
```

步骤05 查看 CIFAR-10 数据。

下载后，我们可以使用下列指令来查看数据项数。

```
In [3]: print('train:',len(x_img_train))
        print('test :',len(x_img_test))

train: 50000
test : 10000
```

从以上执行结果可知，第一维是项数，数据可分为两部分：

- train 训练数据 50 000 项。
- test 测试数据 10 000 项。

9.2 查看训练数据

先查看训练数据。

1. 训练数据是由 images 与 label 所组成的

`y_label_train` 是图像数据的真实值，每一个数字代表一种图像类别的名称，共有 10 个类别：0:airplane、1:automobile、2:bird、3:cat、4:deer、5:dog、6:frog、7:horse、8:ship、9:truck，如图 9-4 所示。

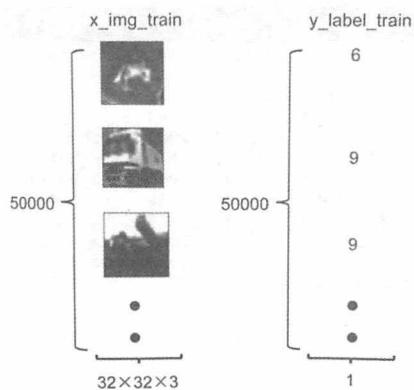


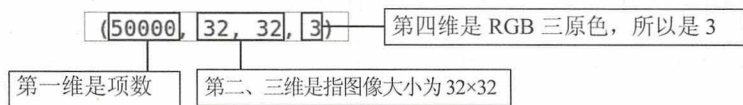
图 9-4

2. Images 的 shape 形状

下面的程序代码使用 `.shape` 方法来查看 `x_img_train` 的 shape。

```
In [4]: x_img_train.shape
Out[4]: (50000, 32, 32, 3)
```

各维的说明如下：



3. 第 0 项 images 图像的内容

下面的程序代码用来查看第 0 项 images 的内容，每一点都是由 RGB 三原色所组成的，RGB 共有 3 个数字，数字的范围从 0 到 255，代表图像的 RGB 颜色。

```
In [7]: x_img_test[0]
Out[7]: array([[158, 112, 49],
               [159, 111, 47],
               [165, 116, 51],
               ...])
```

3 个数字是 RGB 三原色

4. y_label_train 的 shape 形状

```
In [5]: y_label_train.shape
Out[5]: (50000, 1)
```

9.3 查看多项 images 与 label

之前只是显示数据，接下来将修改 `plot_images_labels_prediction()` 函数以显示图像。

步骤01 定义 label_dict 字典。

先以 Python 字典 dict 定义每一个数字所代表的图像类别的名称。

```
In [9]: label_dict={0:"airplane",1:"automobile",2:"bird",3:"cat",4:"deer",
                    5:"dog",6:"frog",7:"horse",8:"ship",9:"truck"}
```

步骤02 修改 plot_images_labels_prediction() 函数。

为了便于查看多项数据 images 与 label，我们将修改第 6 章所创建的 plot_images_labels_prediction()，并且使用 label_dict 字典将 label 与 prediction 的 0~9 数字转换为图像类别名称。

```
In [10]: import matplotlib.pyplot as plt
def plot_images_labels_prediction(images, labels, prediction,
                                  idx, num=10):
    fig = plt.gcf()
    fig.set_size_inches(12, 14)
    if num>25: num=25
    for i in range(0, num):
        ax=plt.subplot(5,5, 1+i)
        ax.imshow(images[idx], cmap='binary')

        title=str(i)+' '+label_dict[labels[i][0]]
        if len(prediction)>0:
            title+='=>' +label_dict[prediction[i]]

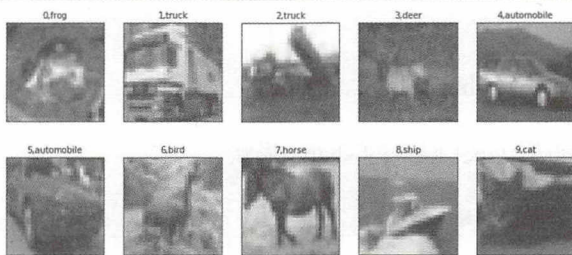
        ax.set_title(title, fontsize=10)
        ax.set_xticks([]);ax.set_yticks([])
        idx+=1
    plt.show()
```

使用 label_dict 转换

步骤03 查看训练数据前 10 项数据。

使用 plot_images_labels_prediction() 显示训练数据前 10 项数据。因为还没有预测数据，所以 prediction 参数输入空的 list[]。

```
In [11]: plot_images_labels_prediction(x_img_train, y_label_train, [], 0)
```



9.4 将 images 进行预处理

为了将 images 的内容送入卷积神经网络模型进行训练与预测，必须先进行数据的预处理。

步骤01 查看训练数据第 1 个图像的第 1 个点。

我们可以用下列指令来查看训练数据的第 1 个图像的第 1 个点。

```
In [14]: x_img_train[0][0][0]
Out[14]: array([59, 62, 63], dtype=uint8)
```

可以看到每一点共有 3 个数字，分别代表 RGB [59,62,63]。

步骤02 将照片图像 image 的数字标准化。

image 的数字标准化可以提高模型的准确率，因为 image 的数字是 0 到 255，所以最简单标准化的方式是除以 255，如下列程序代码：

```
In [15]: x_img_train_normalize = x_img_train.astype('float32') / 255.0
         x_img_test_normalize = x_img_test.astype('float32') / 255.0
```

步骤04 查看照片图像 image 的数字标准化后的结果。

使用下列指令查看照片图像 image 的数字标准化后的结果，全部的数值都在 0 与 1 之间。

```
In [16]: x_img_train_normalize[0][0][0]
Out[16]: array([ 0.23137255,  0.24313726,  0.24705882], dtype=float32)
```

9.5 对 label 进行数据预处理

对于 CIFAR-10 数据集，我们希望预测图像的类型，例如“船”的图像的 label 是 8，经过一位有效编码（One-Hot Encoding）转换为 0000000010，10 个数字正好对应输出层 10 个神经元。

步骤01 查看 label 原来的 shape 形状。

使用下列指令来查看 label 原来的 shape 形状。

```
In [17]: y_label_train.shape
Out[17]: (50000, 1)
```

我们可以看到以上执行的结果共计 50 000 项，每一项是 1 个 0~9 之间的数字。

步骤02 查看前 5 项数据。

使用下列指令来查看前 5 项数据：我们可以看到都是 0~9 的数字，代表图像的分类。

```
In [18]: y_label_train[:5]
Out[18]: array([[6],
                [9],
                [9],
                [4],
                [1]], dtype=uint8)
```

步骤03 将 label 标签字段转换为一位有效编码 (One-Hot Encoding)。

Keras 提供了 `np_utils.to_categorical` 方法, 可以进行 One-Hot Encoding 转换。下面的程序代码将训练数据与测试数据的 label 标签字段进行 One-Hot Encoding 转换。

```
In [19]: from keras.utils import np_utils
         y_label_train_OneHot = np_utils.to_categorical(y_label_train)
         y_label_test_OneHot = np_utils.to_categorical(y_label_test)
```

步骤04 One-Hot Encoding 转换之后的 label 标签字段。

```
In [20]: y_label_train_OneHot.shape
Out[20]: (50000, 10)
```

从以上执行结果可知, 共计 50 000 项, 每一笔是 10 个 0 或 1 的组合。

步骤05 查看转换为 One-Hot Encoding 之后的结果。

```
In [21]: y_label_train_OneHot[:5]
Out[21]: array([[0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
                [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

查看以上执行结果可知, 第 1 项数据原来的真实值是 6, 执行 One-Hot Encoding 转换后变成 0 或 1 的组合, 只有第 6 个数字 (从 0 算起) 是 1, 其余都是 0。

9.6 结论

本章介绍了下载并且读取 CIFAR-10 数据集, 还介绍了 CIFAR-10 数据集的特色, 并且已经完成数据的预处理。在下一章, 我们可以使用 Keras 建立卷积神经网络模型, 训练模型并进行预测。

第10章

Keras卷积神经网络 识别CIFAR-10图像

在本章中，我们将介绍使用 Keras 建立卷积神经网络模型，并且训练模型、评估模型准确率，然后使用训练完成的模型来识别 CIFAR-10 图像数据集。由于 CIFAR-10 图像识别的难度比 MNIST 数据集识别的难度高很多，因此我们尝试以更多次的卷积与池化运算来提高识别的准确率。

10.1 卷积神经网络简介

步骤01 卷积神经网络介绍

我们将建立的卷积神经网络如图 10-1 所示。

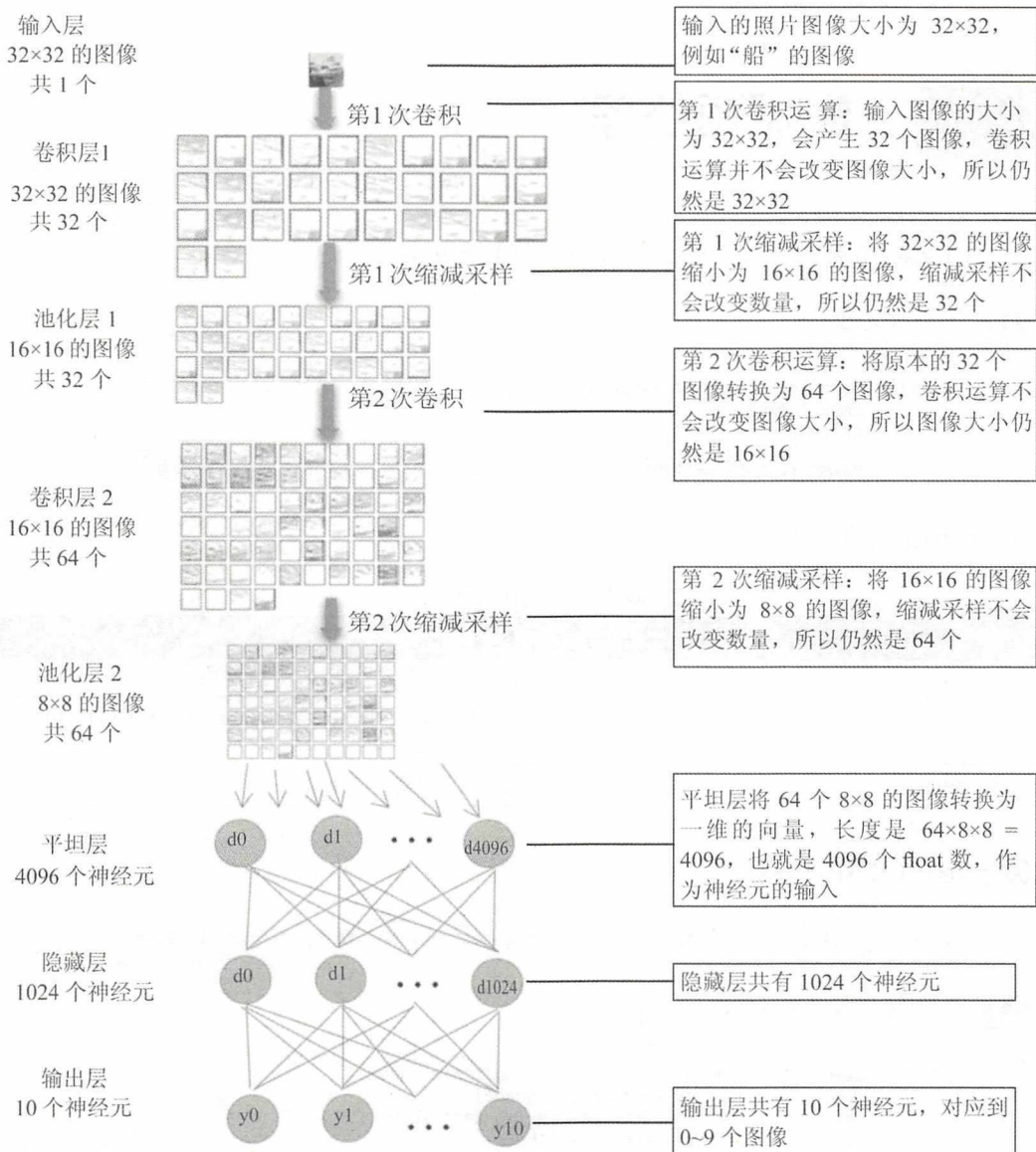


图 10-1

如图 10-1 所示，我们可以看到卷积神经网络可分为以下两大部分。

- **图像的特征提取**：通过卷积层 1、池化层 1、卷积层 2、池化层 2 的处理，提取图像的特征。
- **完全连接神经网络（fully connected layer）**：包含平坦层、隐藏层、输出层所组成的类神经网络。

本章完整的程序代码可参考范例程序 Keras_Cifar_CNN. ipynb。范例程序下载与安装可参考本书附录 A。

10.2 数据预处理

关于 CIFAR-10 数据预处理的详细说明可参考第 9 章。

步骤01 导入所需模块。

```
In [1]: from keras.datasets import cifar10
import numpy as np
np.random.seed(10)

Using TensorFlow backend.
```

程序代码说明见表 10-1。

表 10-1 程序代码说明

程序代码	说明
from keras.datasets import cifar10	从 keras.datasets 导入 CIFAR-10 数据集
import numpy as np	导入 NumPy 模块，NumPy 是 Python 语言的扩展程序库，支持维度数组与矩阵运算
np.random.seed(10)	设置 seed 产生需要的随机数

步骤02 读取 CIFAR-10 数据。

```
In [2]: (x_img_train,y_label_train),(x_img_test,y_label_test)=cifar10.load_data()
```

步骤03 显示训练与验证数据的 shape。

```
In [3]: print("train data:",'images:',x_img_train.shape,
" labels:",y_label_train.shape)
print("test data:",'images:',x_img_test.shape ,
" labels:",y_label_test.shape)

train data: images: (50000, 32, 32, 3) labels: (50000, 1)
test data: images: (10000, 32, 32, 3) labels: (10000, 1)
```




步骤04 将 features（照片图像特征值）标准化。

将 features（照片图像特征值）标准化可以提高模型预测的准确度，并且更快收敛。

```
In [4]: x_img_train_normalize = x_img_train.astype('float32') / 255.0  
        x_img_test_normalize = x_img_test.astype('float32') / 255.0
```

步骤05 label（照片图像真实的值）以一位有效编码进行转换。

以下程序代码将训练数据与测试数据的 label 进行一位有效编码转换。

```
In [5]: from keras.utils import np_utils  
        y_label_train_OneHot = np_utils.to_categorical(y_label_train)  
        y_label_test_OneHot = np_utils.to_categorical(y_label_test)
```

10.3 建立模型

我们将使用下面的程序代码建立卷积神经网络。

1. 导入所需模块

```
In [7]: from keras.models import Sequential  
        from keras.layers import Dense, Dropout, Activation, Flatten  
        from keras.layers import Conv2D, MaxPooling2D, ZeroPadding2D
```

➤ 导入 keras 的 Sequential 模块

```
from keras.models import Sequential
```

➤ 导入 keras 的 layers 模块

```
from keras.layers import Dense, Dropout, Activation, Flatten
```

➤ 导入 keras 的 layers 模块

```
from keras.layers import Conv2D, MaxPooling2D, ZeroPadding2D
```

我们将使用下面的程序代码建立卷积神经网络，如图 10-2 所示。

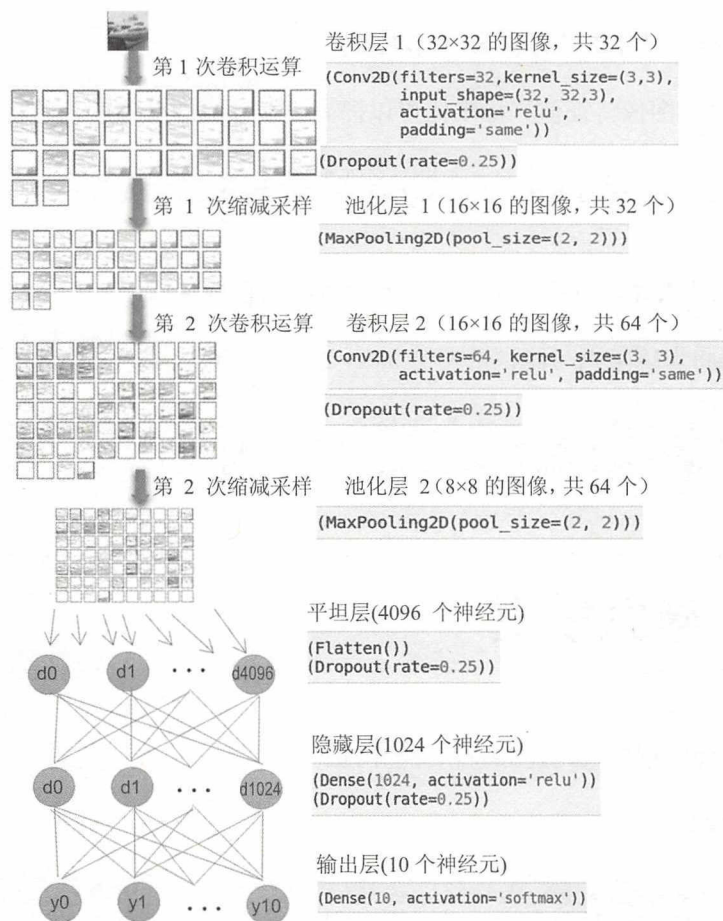


图 10-2

2. 建立 keras 的 Sequential 模型

下面的程序代码建立一个 Sequential 线性堆叠模型, 后续只需要将各个神经网络层加入模型即可。

```
In [7]: model = Sequential()
```

3. 建立卷积层 1 与池化层 1

一个完整的卷积运算包含一个卷积层与一个池化层。

➤ 建立卷积层 1

使用下列程序代码建立卷积层 1。输入图像的大小为 32×32, 会产生 32 个图像, 卷积运算并不会改变图像大小, 所以图像大小仍然是 32×32。卷积运算的效果很类似滤镜效果, 可以提取图像的特征。



```
In [10]: model.add(Conv2D(filters=32, kernel_size=(3,3),
                        input_shape=(32, 32,3),
                        activation='relu',
                        padding='same'))
```

以上程序代码把 Conv2D 层加入模型中，需输入表 10-2 中的参数。

表 10-2 把 Conv2D 层加入模型中需输入的参数

程序代码	说明
<code>filters=32,</code>	设置随机产生 32 个滤镜
<code>kernel_size=(3,3),</code>	每一个滤镜大小为 3×3
<code>padding='same',</code>	这个设置让卷积运算产生的卷积图像大小不变
<code>input_shape=(32, 32,3),</code>	第 1、2 维：代表输入的图像形状大小为 32×32，第 3 维：因为是彩色，所以代表 RGB 三原色是 3
<code>activation='relu'</code>	设置 ReLU 激活函数

➤ 加入 Dropout 避免过度拟合

以下程序代码把 Dropout 加入模型中，Dropout(0.25)的功能是，每次训练迭代时，会随机地在神经网络中放弃 25%的神经元，以避免过度拟合。

```
In [11]: model.add(Dropout(rate=0.25))
```

➤ 建立池化层 1

下面的程序代码建立池化层 1，输入参数 `pool_size=(2,2)`，执行第一次缩减采样，将 32×32 的图像缩小为 16×16 的图像，缩减采样不会改变数量，所以仍然是 32 个。

```
In [12]: model.add(MaxPooling2D(pool_size=(2, 2)))
```

4. 建立卷积层 2 与池化层 2

➤ 建立卷积层 2

使用下面的程序代码建立卷积层 2。执行第 2 次卷积运算：将原本的 32 个图像转换为 64 个图像，卷积运算不会改变图像大小，所以图像大小仍然是 16×16。

```
In [14]: model.add(Conv2D(filters=64, kernel_size=(3, 3),
                        activation='relu', padding='same'))
```

以上程序代码把 Conv2D 层加入模型中，需输入表 10-3 中的参数。

表 10-3 把 Conv2D 层加入模型中需输入的参数

程序代码	说明
<code>filters=64</code>	创建 64 个滤镜
<code>kernel_size=(3,3),</code>	每一个滤镜大小为 3×3
<code>padding='same',</code>	这个设置让卷积运算产生的图像大小不变
<code>activation='relu'</code>	设置 ReLU 激活函数

➤ 加入 Dropout 避免过度拟合

以下程序代码加入 Dropout(0.25)的功能是，每次训练迭代时会随机地在神经网络中放弃 25%的神经元，以避免过度拟合。

```
In [15]: model.add(Dropout(0.25))
```

➤ 建立池化层 2

以下程序代码建立池化层 2，输入参数 pool_size=(2,2)，执行第二次缩减采样，将 16×16 的图像缩小为 8×8 的图像，缩减采样不会改变数量，所以仍然是 64 个。

```
In [16]: model.add(MaxPooling2D(pool_size=(2, 2)))
```

5. 建立神经网络（平坦层、隐藏层、输出层）

➤ 建立平坦层

下面的程序代码建立平坦层，将前面的步骤建立的“池化层 2”的 64 个 8×8 的图像转换为一维的向量，长度是 64×8×8=4096，也就是 4096 个 Float 数，正好对应 4096 个神经元。并且加入 Dropout(0.25)，每次训练迭代时，会随机地在神经网络放弃 25%的神经元，以避免过度拟合。

```
In [18]: model.add(Flatten())  
         model.add(Dropout(rate=0.25))
```

➤ 建立隐藏层

下面的程序代码建立隐藏层，共有 1024 个神经元，并且加入 Dropout(0.25)，随机去掉 25%的神经元，以避免过度拟合。

```
In [19]: model.add(Dense(1024, activation='relu'))  
         model.add(Dropout(rate=0.25))
```

➤ 建立输出层

最后建立输出层，共有 10 个神经元，对应 0~9 共 10 个图像类别。并且使用 softmax 激活函数进行转换，softmax 可以将神经元的输出转换为预测每一个图像类别的概率。

```
In [20]: model.add(Dense(10, activation='softmax'))
```

6. 查看模型的摘要

可以使用下列指令来查看模型的摘要。

```
In [21]: print(model.summary())
```

模型的摘要如图 10-3 所示。



Layer (type)	Output Shape	Param #	
conv2d_1 (Conv2D)	(None, 32, 32, 32)	896	卷积层 1 与池化层 1
dropout_1 (Dropout)	(None, 32, 32, 32)	0	
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 32)	0	
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18496	卷积层 2 与池化层 2
dropout_2 (Dropout)	(None, 16, 16, 64)	0	
max_pooling2d_2 (MaxPooling2D)	(None, 8, 8, 64)	0	
flatten_1 (Flatten)	(None, 4096)	0	神经网络（平坦层、隐藏层、输出层）
dropout_3 (Dropout)	(None, 4096)	0	
dense_1 (Dense)	(None, 1024)	4195328	
dropout_4 (Dropout)	(None, 1024)	0	
dense_2 (Dense)	(None, 10)	10250	
Total params: 4,224,970.0			
Trainable params: 4,224,970.0			
Non-trainable params: 0.0			

图 10-3

10.4 进行训练

当我们建立好深度学习模型后，就可以使用反向传播算法进行训练。详细说明可参考第 2 章。

1. 定义训练方式

在训练模型之前，我们必须使用 `compile` 方法对训练模型进行设置，如下列指令：

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])
```

`compile` 方法需输入 3 个参数：`loss`、`optimizer` 和 `metrics`（对这 3 个参数的解释说明可参考 7.4 节）。

2. 开始训练

```
In [24]: train_history=model.fit(x_img_train_normalize, y_label_train_OneHot,
                                validation_split=0.2,
                                epochs=10, batch_size=128, verbose=1)
```

以上程序代码说明如下：

使用 `model.fit` 进行训练，训练过程会存储在 `train_history` 变量中，需输入下列参数。

(1) 输入训练数据参数

- `x=x_img_train_normalize` (features 照片图像的特征值, 经过标准化处理)。
- `y=y_label_train_OneHot` (label 照片图像真实的值, 经过 One-Hot Encoding 转换)。

(2) 设置训练与验证数据比例

- 设置参数 `validation_split=0.2`。

训练之前 Keras 会自动将数据分成: 80%作为训练数据, 20%作为验证数据。因为总共 50 000 项数据, 所以分成: $50\,000 \times 0.8 = 40\,000$ 项作为训练数据, $50\,000 \times 0.2 = 10\,000$ 项作为验证数据, 如图 10-4 所示。

(3) 设置训练周期次数与每一批次项数

- `epochs=10`: 执行 10 个训练周期。
- `batch_size=128`: 每一批次 128 项数据。

(4) 设置显示训练过程

- `verbose=2`: 显示训练过程。

以上程序代码共执行了 10 个训练周期, 每一个训练周期都执行下列功能:

- 使用 40 000 项训练数据进行训练, 分为每一批次 128 项, 所以大约分为 160 批次 ($40\,000/128=313$) 进行训练。
- Epoch (训练周期) 训练完成后, 会计算这个训练周期的准确率与误差, 并且在 `train_history` 中新增一项数据记录。

执行结果如图 10-4 所示。

Train on 40000 samples, validate on 10000 samples		80% 作为训练数据, 20% 作为验证数据				
Epoch 1/20	40000/40000	[=====]	- 155s	- loss: 1.5049	- acc: 0.4583	- val_loss: 1.2913 - val_acc: 0.5759
Epoch 2/20	40000/40000	[=====]	- 155s	- loss: 1.1460	- acc: 0.5942	- val_loss: 1.1144 - val_acc: 0.6374
Epoch 3/20	40000/40000	[=====]	- 161s	- loss: 0.9900	- acc: 0.6517	- val_loss: 1.0083 - val_acc: 0.6686
Epoch 4/20	40000/40000	[=====]	- 161s	- loss: 0.8873	- acc: 0.6878	- val_loss: 0.9595 - val_acc: 0.6920
Epoch 15/20	40000/40000	[=====]	- 166s	- loss: 0.2466	- acc: 0.9141	- val_loss: 0.7823 - val_acc: 0.7397
Epoch 16/20	40000/40000	[=====]	- 166s	- loss: 0.2224	- acc: 0.9215	- val_loss: 0.7616 - val_acc: 0.7534
Epoch 17/20	40000/40000	[=====]	- 167s	- loss: 0.2111	- acc: 0.9266	- val_loss: 0.8185 - val_acc: 0.7405
Epoch 18/20	40000/40000	[=====]	- 168s	- loss: 0.1892	- acc: 0.9345	- val_loss: 0.8001 - val_acc: 0.7456
Epoch 19/20	40000/40000	[=====]	- 168s	- loss: 0.1753	- acc: 0.9394	- val_loss: 0.8479 - val_acc: 0.7369
Epoch 20/20	40000/40000	[=====]	- 167s	- loss: 0.1686	- acc: 0.9417	- val_loss: 0.8197 - val_acc: 0.7450

图 10-4

从以上执行结果的屏幕显示界面中, 我们可以看到共执行了 10 个训练周期, 同时可以发

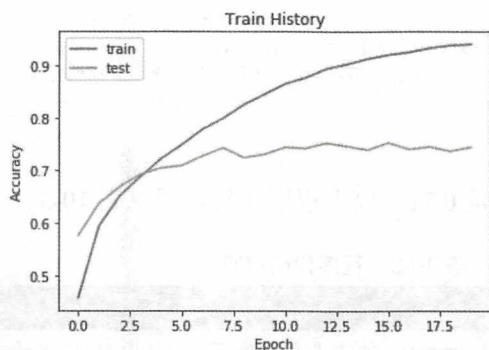


现误差越来越小，准确率越来越高。

3. 画出准确率执行的结果

下面是程序代码画出准确率执行的结果。有关 `show_train_history` 可参考第 7 章的说明。

```
In [26]: show_train_history('acc','val_acc')
```

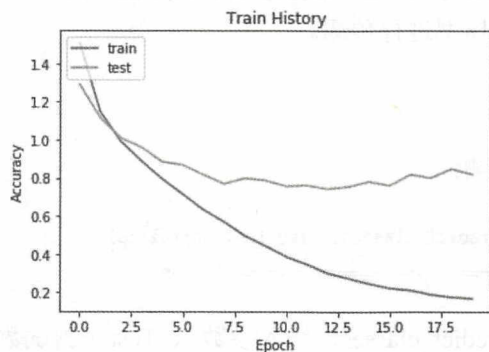


在以上执行结果的屏幕显示界面中，“acc 训练的准确率”是深色的，“val_acc 验证的准确率”是浅色的，总共执行了 10 个训练周期，我们可以发现：

- 无论是训练还是验证，准确率都越来越高。
- 在 Epoch 训练后期，“acc 训练的准确率”比“val_acc 验证的准确率”高。

4. 画出误差的执行结果

```
In [27]: show_train_history('loss','val_loss')
```



在以上执行结果的屏幕显示界面中，“loss 训练的误差”是深色的，“val_loss 验证的误差”是浅色的，总共执行了 10 个训练周期，我们可以发现：

- 无论是训练还是验证，验证的误差都越来越低。
- 在 Epoch 训练后期，“loss 训练的误差”比“val_loss 验证的误差”小。

10.5 评估模型准确率

之前我们已经完成训练，现在要使用 test 测试数据集，评估模型准确率。用下面的程序代码来评估模型准确率。

```
In [28]: scores = model.evaluate(x_img_test_normalize,
                                y_label_test_OneHot, verbose=0)
          scores[1]
Out[28]: 0.71140000000000003
```

从以上的执行结果可知准确率是 0.71。以上程序代码说明见表 10-3。

表 10-3 程序代码说明

程序代码	说明
scores=model.evaluate	使用 model.evaluate 评估模型的准确率，评估后的准确率会存储在 scores 中
x_img_test_normalize	测试数据的 features（照片图像的特征值，经过标准化处理）
y_label_test_OneHot	测试数据的 label（照片图像真实的值，经过一位有效编码转换

10.6 进行预测

恭喜你，之前的步骤我们建立好了模型，并且完成了模型的训练，准确率达到还可以接受的 0.97，接下来我们将使用此模型进行预测。

1. 执行预测

我们可以使用下列指令执行预测：

```
In [29]: prediction=model.predict_classes(x_img_test_normalize)
          10000/10000 [=====] - 6s
```

下面的程序代码使用 model.predict_classes，输入参数 x_Test（测试数据的照片图像）来进行预测。

2. 预测结果

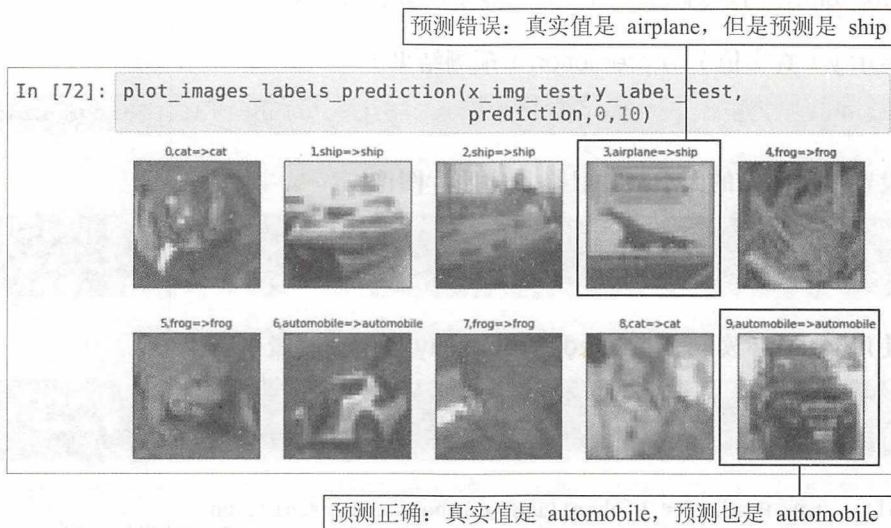
可以使用下列指令来查看预测结果的前 10 项数据：

```
In [30]: prediction[:10]
Out[30]: array([3, 8, 8, 8, 6, 6, 1, 6, 3, 1], dtype=int64)
```

可以看到第 1 项预测的结果是 3，第 2 项是 8……

3. 显示前 10 项预测结果

使用第 9 章修改的 `plot_images_labels_prediction` 函数显示前 10 项预测结果，传入测试数据图像、label（真实值）及 prediction（预测结果）。



10.7 查看预测概率

有时我们不只希望知道预测结果，还想要知道预测每一种类别的概率。

1. 使用测试数据进行预测

我们使用 `model.predict` 输入测试数据，就可以预测概率。

```
In [33]: Predicted_Probability=model.predict(x_img_test_normalize)
```

2. 建立 `show_Predicted_Probability` 函数

```
In [50]: def show_Predicted_Probability(y,prediction,
      x_img,Predicted_Probability,i):
    print('label:',label_dict[y[i][0]],
          'predict',label_dict[prediction[i]])
    plt.figure(figsize=(2,2))
    plt.imshow(np.reshape(x_img_test[i],(32, 32,3)))
    plt.show()
    for j in range(10):
        print(label_dict[j]+
              ' Probability:%1.9f'%(Predicted_Probability[i][j]))
```


➤ 定义 show_Predicted_Probability 函数

```
def show_Predicted_Probability(y,prediction,
                               x_img,Predicted_Probability,i):
```

传入参数：y（真实值）、prediction（预测结果）、x_img（预测的图像）、Predicted_Probability（预测概率）、i（开始显示的数据 index）。

➤ 显示 y（真实值）与 prediction（预测结果）

```
print('label:',label_dict[y[i][0]], 'predict:',label_dict[prediction[i]])
```

➤ 设置显示图像的大小，并且显示出照片图像

```
plt.figure(figsize=(2,2)) plt.imshow(np.reshape(x_img_test[i],(32, 32,3)))
plt.show()
```

➤ 使用 for 循环读取 Predicted_Probability 显示预测概率

```
for j in range(10): print(label_dict[j]+
' Probability:%1.9f'%(Predicted_Probability[i][j]))
```

```
In [51]: show_Predicted_Probability(y_label_test,prediction,
                                     x_img_test,Predicted_Probability,0)
label: cat predict cat
```

3. 查看第 0 项数据预测的概率（见图 10-5）

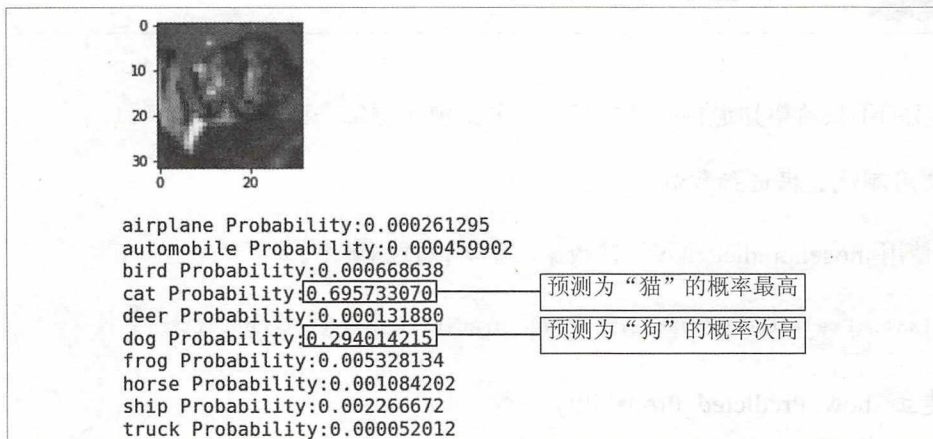


图 10-5

从以上执行结果可知，这张照片图像预测为“猫”的概率最高，预测为“狗”的概率次高，所以最后预测的结果是“猫”预测正确。



4. 查看第 3 项数据预测的概率（见图 10-6）

```
In [52]: show_Predicted_Probability(y_label_test,prediction,
                                     x_img_test,Predicted_Probability,3)
```

label: airplane predict ship

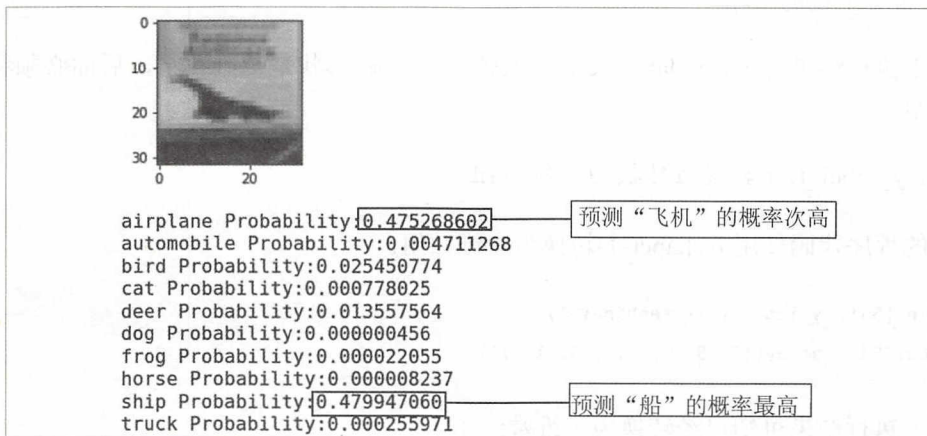


图 10-6

从以上执行结果可知，这张照片图像预测为“船”的概率最高，预测为“飞机”的概率次高，所以最后预测的结果是“船”。但是真实值是“飞机”，所以此项预测是错误的。

10.8 显示混淆矩阵

在上一节中我们看到了一个预测错误：真实值“飞机”预测成“船”了。如果我们想要进一步知道所建立的模型中哪些图像类别的预测准确率最高，哪些图像类别最容易混淆（预测错误），就可以使用混淆矩阵来显示。

1. 查看预测结果的形状

我们将使用 `pd.crosstab` 建立混淆矩阵，但是 `pd.crosstab` 的输入都必须是一维数组，所以要先确认 `prediction`（预测结果）与 `y_label_test`（真实值）是一维数组。如果不是一维数组，就必须先转换为一维数组。下列指令用于查看预测结果的 `shape` 形状。

```
In [37]: prediction.shape
```

```
Out[37]: (10000,)
```

从以上执行结果可知预测结果是一维数组。

2. 查看 y_label_test 真实值的 shape 形状

用下列指令来查看 y_label_test 真实值的 shape 形状。

```
In [55]: y_label_test.shape
Out[55]: (10000, 1)
```

从以上执行结果可知，y_label_test 真实值的 shape 形状是二维数组，后面必须将它转换为一维数组。

3. 将 y_label_test 真实值转换为一维数组

下面的程序代码使用 reshape(-1)转换为一维数组。

```
In [53]: y_label_test.reshape(-1)
Out[53]: array([3, 8, 8, ..., 5, 1, 7])
```

从以上执行结果可知已经转换为 1 维数组了。

4. 使用 pandas crosstab 建立混淆矩阵

Pandas 提供了用于建立混淆矩阵的功能。

```
In [41]: import pandas as pd
print(label_dict)
pd.crosstab(y_label_test.reshape(-1),prediction,
            rownames=['label'],colnames=['predict'])
```

程序代码说明见表 10-4。

表 10-4 程序代码说明

程序代码	说明
import pandas as pd	导入 pandas 模块，后续会以 pd 来引用
print(label_dict)	显示 label_dict 字典，方便后续对照查看
pd.crosstab(y_label_test.reshape(-1), prediction, rownames=['label'], colnames=['predict'])	使用 pd.crosstab 建立混淆矩阵，输入下列参数： 测试数据的真实值，使用.reshape(-1)转换为一维数组 测试数据的预测结果 设置行的名称是 label 设置列的名称是 predict

执行后显示混淆矩阵，如图 10-7 所示。

```
{0: 'airplane', 1: 'automobile', 2: 'bird', 3: 'cat', 4: 'deer'
, 5: 'dog', 6: 'frog', 7: 'horse', 8: 'ship', 9: 'truck'}
```


predict	0	1	2	3	4	5	6	7	8	9
label										
0	771	13	57	16	16	6	13	8	75	25
1	10	860	10	16	4	7	18	3	17	55
2	53	3	681	47	64	49	65	20	10	8
3	17	9	79	555	52	158	93	25	4	8
4	14	1	103	63	669	33	61	43	11	2
5	13	4	60	184	41	601	53	33	5	6
6	3	4	52	41	15	11	868	1	4	1
7	10	1	35	28	71	53	12	781	4	5
8	40	31	16	20	13	7	6	0	848	19
9	32	98	18	25	4	8	12	14	33	756

真实值是 3，但是预测是 5

对角线是预测正确的数字

真实值是 5 “狗”，但是预测是 3 “猫”

图 10-7

从以上混淆矩阵，我们观察如下。

● 对角线是预测正确的，我们发现：

- 真实值是 6 “蛙”，被正确预测为 6 “蛙” 的项数有 868 项，最高，最不容易混淆。
- 真实值是 3 “猫”，被正确预测为 3 “猫” 的项数有 555 项，最低，也就是说最容易混淆。

● 其他非对角线的数字代表将某一个标签错误预测成为另一个标签，最容易混淆：

- 真实值是 5 “狗”，但预测是 3 “猫”，项数有 184 项，最高，也就是说 “狗” 很容易被误认为 “猫”。
- 真实值是 3 “猫”，但预测是 5 “狗”，项数有 158 项，次高，也就是说 “狗” 很容易被误认为 “猫”。人类识别也很容易搞错 “狗” 与 “猫”，难怪机器学习也预测错误。

➤ 动物类不容易混淆为交通工具类

CIFAR-10 的图像类别大约可以分为以下两大类。

- 动物类：2（鸟）、3（猫）、4（鹿）、5（狗）、6（蛙）、7（马）。
- 交通工具类：0（飞机）、1（汽车）、8（船）、9（卡车）。

我们还发现了一些有趣的现象：

- 2、3、4、5、6、7 预测为 1 的数量都是个位数。

也就是说，2（鸟）、3（猫）、4（鹿）、5（狗）、6（蛙）、7（马）都是动物，不容易混淆为 1（汽车）。

- 但是 8、9 预测为 1 的数量都分别是十位数。

也就是说，8（船）、9（卡车）属于交通工具，容易混淆为 1（汽车），如图 10-8 所示。

predict	0	1	2	3	4	5	6	7	8	9
label										
0	771	13	57	16	16	6	13	8	75	25
1	10	860	10	16	4	7	18	3	17	55
2	53	3	681	47	64	49	65	20	10	8
3	17	9	79	555	52	158	93	25	4	8
4	14	1	103	63	669	33	61	43	11	2
5	13	4	60	184	41	601	53	33	5	6
6	3	4	52	41	15	11	868	1	4	1
7	10	1	35	28	71	53	12	781	4	5
8	40	31	16	20	13	7	6	0	848	19
9	32	98	18	25	4	8	12	14	33	756

2（鸟）、3（猫）、4（鹿）、5（狗）、6（蛙）、7（马）都是动物，不容易混淆为1（汽车）

8（船）、9（卡车）属于交通工具，容易混淆为1（汽车）

图 10-8

10.9 建立 3 次的卷积运算神经网络

之前建立的卷积式神经网络执行结果的准确率只有 0.738，我们希望能通过更多次的卷积运算提高准确率。下面的程序代码可参考范例程序 Keras_Cifar_CNN_Deeper_Conv3.ipynb。

1. 建立 3 次的卷积运算的神经网络架构图（见图 10-9）
2. 建立卷积层 1 与池化层 1

在下面的程序代码中，我们增加了一次 Conv2D 卷积运算。

```
In [10]: model.add(Conv2D(filters=32, kernel_size=(3, 3), input_shape=(32, 32, 3),
                        activation='relu', padding='same'))
        model.add(Dropout(0.3))
        model.add(Conv2D(filters=32, kernel_size=(3, 3),
                        activation='relu', padding='same'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
```

增加Conv2D 层

3. 建立卷积层 2 与池化层 2

```
In [12]: model.add(Conv2D(filters=64, kernel_size=(3, 3),
                        activation='relu', padding='same'))
        model.add(Dropout(0.3))
        model.add(Conv2D(filters=64, kernel_size=(3, 3),
                        activation='relu', padding='same'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
```

增加Conv2D 层

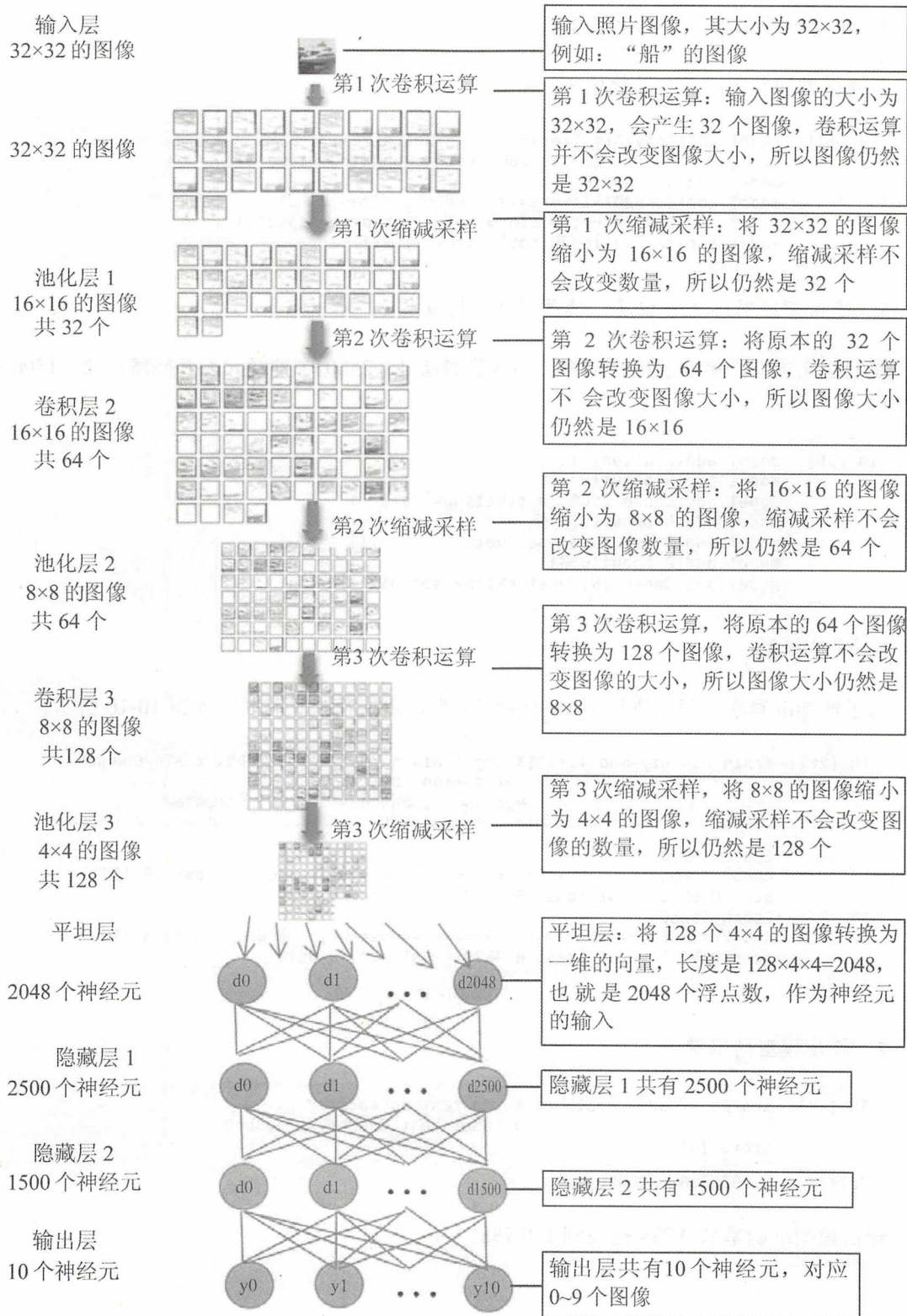


图 10-9

4. 新增加卷积层 3 与池化层 3

下面再增加卷积层 3 与池化层 3。

```
In [14]: model.add(Conv2D(filters=128, kernel_size=(3, 3),
                        activation='relu', padding='same'))
model.add(Dropout(0.3))
model.add(Conv2D(filters=128, kernel_size=(3, 3),
                        activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

5. 建立神经网络（平坦层、隐藏层 1、隐藏层 2、输出层）

我们建立更宽、更深的神经网络，加入隐藏层 1（2500 个神经元）和隐藏层 2（1500 个神经元）。

```
In [16]: model.add(Flatten())
model.add(Dropout(0.3))
model.add(Dense(2500, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1500, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(10, activation='softmax'))
```

6. 训练模型

为了增加准确率，我们执行了 50 个训练周期，需要花很长时间，如图 10-10 所示。

```
In [20]: train_history=model.fit(X_img_train_normalize, y_label_train_OneHot,
                        validation_split=0.2,
                        epochs=50, batch_size=300, verbose=1)
```

```
Epoch 49/50
40000/40000 [=====] - 360s - loss: 0.1373 -
acc: 0.9518 - val_loss: 0.7770 - val_acc: 0.7929
Epoch 50/50
40000/40000 [=====] - 358s - loss: 0.1406 -
acc: 0.9517 - val_loss: 0.7374 - val_acc: 0.8016
```

图 10-10

7. 评估模型的准确率

```
In [24]: scores = model.evaluate(X_img_test_normalize,
                        y_label_test_OneHot, verbose=0)
scores[1]
```

```
Out[24]: 0.78890000000000005
```

评估模型准确率从 0.738 提高到了 0.788。

10.10 模型的保存与加载

上一节的 Keras_Cifar_CNN_Deeper_Conv3.ipynb 程序训练必须花很长时间，往往需要数小时。有时还可能因为某些原因导致计算机宕机，这样之前的训练就前功尽弃了，解决的方法是：每次程序执行完成训练后，将模型权重保存一下。下次程序执行训练之前，先加载模型权重，再继续训练。

下面的程序代码可参考范例程序 Keras_Cifar_CNN_Continue_Train.ipynb。

步骤01 设置训练周期。

每次训练周期不要设置得太多，例如下面的范例中，epochs 设置为 5。

```
In [24]: train_history=model.fit(x_img_train_normalize, y_label_train_OneHot,
                                validation_split=0.2,
                                epochs=5, batch_size=128, verbose=1)
```

步骤02 在执行训练之前加载模型权重。

在执行训练之前，先使用 model.load_weights 加载模型权重。

```
In [22]: try:
          model.load_weights("SaveModel/cifarCnnModel.h5")
          print("加载模型成功！继续训练模型")
        except :
          print("加载模型失败！开始训练一个新模型")
```

加载模型失败！开始训练一个新模型

从以上执行结果可知，因为第一次执行尚未保存模型权重，所以会显示“加载模型失败！开始训练一个新模型”。

步骤03 在程序的最后保存模型权重。

将这次执行 5 个训练周期的结果使用 model.save_weights 保存在文件中。

```
In [44]: model.save_weights("SaveModel/cifarCnnModel.h5")
          print("Saved model to disk")

Saved model to disk
```

步骤04 第 2 次执行程序。

第 2 次执行程序，在执行训练之前，一样先使用 model.load_weights 加载模型权重。



```
In [22]: try:
          model.load_weights("SaveModel/cifarCnnModel.h5")
          print("加载模型成功！继续训练模型")
        except:
          print("加载模型失败！开始训练一个新模型")

          加载模型成功！继续训练模型
```

从以上执行结果可知，因为第 2 次执行会加载之前保存的模型权重，所以会显示“加载模型成功！继续训练模型”，这样就可以接在第一次的执行结果之后继续训练。

10.11 结论

在本章中，我们介绍了使用 Keras 建立卷积神经网络识别 CIFAR-10 图像数据。后续章节我们将以多层感知器模型来预测泰坦尼克号乘客的生存概率。

第11章

Keras泰坦尼克号上的 旅客数据集

泰坦尼克号的沉没是历史悲剧。1912 年 4 月 15 日泰坦尼克号在首航时，撞上冰山沉没，乘客和船员共 2224 人，其中 1502 人死亡。这场悲剧震撼了国际社会，之后船舶行业制定了更好的安全规定。泰坦尼克号的旅客数据集被完整地保留下来。我们在本章中先介绍数据的预处理，下一章将建立多层感知器模型来预测每一位乘客的存活率。

如图 11-1 所示，以多层感知器模型来预测泰坦尼克号乘客的生存概率，可分为训练与预测两部分。

➤ 训练

泰坦尼克号数据集的训练数据共有 1309 项，经过数据预处理后会产生 feature，（共有 9 个特征字段，例如性别、年龄等）与 label 标签字段（是否生存？1：是，2：否），然后输入多层感知器模型进行训练，训练完成的模型就可以在下一阶段进行预测时使用。

➤ 预测

输入新的泰坦尼克号数据，预处理后会产生 features（9 个特征字段），使用训练完成的多层感知器模型进行预测，最后产生预测结果：生存概率。

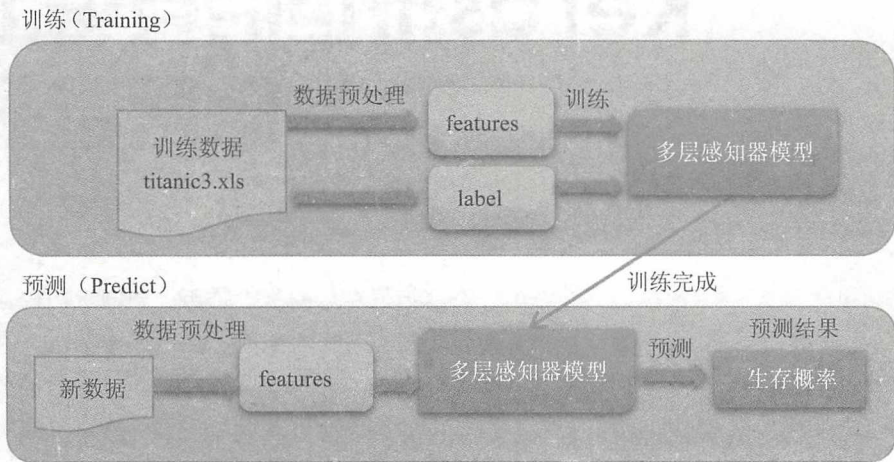


图 11-1

本章完整的程序代码可参考范例程序 Keras-Taianic_Introduce.ipynb。范例程序下载与安装可参考本书附录 A。

11.1 下载泰坦尼克号旅客数据集

1. 导入下载所需模块

```
In [1]: import urllib.request
import os
```

以上程序代码的说明见表 11-1。

表 11-1 程序代码说明

程序代码	说明
import urllib.request	导入 urllib 模块，将用于下载文件
import os	导入 os 模块，用于确认文件是否存在

2. 下载泰坦尼克号的旅客数据集

使用下面的程序代码来下载泰坦尼克号的旅客数据集。

```
In [2]: url="http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic3.xls"
        filepath="data/titanic3.xls"
        if not os.path.isfile(filepath):
            result=urllib.request.urlretrieve(url,filepath)
            print('downloaded:',result)

download: ('data/titanic3.xls', <http.client.HTTPMessage object at 0x7f20e4...
```

从以上执行结果可知，程序会下载 titanic3.xls，并且存储在程序执行目录下的 data 目录中。

➤ 设置下载的网址

```
url="http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic3.xls"
```

➤ 设置存储文件的路径

```
filepath="data/titanic3.xls"
```

➤ 判断文件不存在就会下载文件

```
if not os.path.isfile(filepath):
    result=urllib.request.urlretrieve(url,filepath) print('downloaded:',result)
```

以上程序代码在判断文件不存在之后，就会使用 urllib.request.urlretrieve 下载文件。输入参数：url（下载的网址）与 filepath（存储文件的路径）。

3. 查看已下载的文件

查看下载的数据文件，根据使用的环境是 Windows 或 Linux Ubuntu 而稍有不同，说明如下：

➤ 在 Windows 下查看已下载的数据文件

我们可以使用文件资源管理器来查看程序执行目录下的 data 目录，例如：我们的执行目录是 C:\pythonwork\keras，就可以在 C:\pythonwork\keras\data 看到已下载的 titanic3.xls 文件，如图 11-2 所示。



图 11-2

➤ 在 Linux Ubuntu 下查看已下载的数据文件

在“终端”程序输入下列命令，先切换到程序执行目录，再查看目录。


```
cd ~/pywork/keras/data ll titanic3.xls
```

执行后屏幕显示的界面如图 11-3 所示，我们可以看到已下载的 titanic3.xls 文件。

```
user@Ubuntu1604: ~/pywork/keras/data
user@Ubuntu1604:~$ cd ~/pywork/keras/data
user@Ubuntu1604:~/pywork/keras/data$ ll titanic3.xls
-rwxrwxrwx 1 root root 284160 3月 10 11:08 titanic3.xls*
user@Ubuntu1604:~/pywork/keras/data$
```

图 11-3

11.2 使用 Pandas DataFrame 读取数据并进行预处理

Pandas 为 Python 提供了 DataFrame 功能，可以很方便地处理数据。

步骤01 导入所需模块。

```
In [3]: import numpy
import pandas as pd
```

步骤02 读取 titanic3.xls。

下面的程序代码使用 Pandas 所提供的 read_excel()方法把 titanic3.xls 文件读取到 all_df DataFrame。

```
In [4]: all_df = pd.read_excel(filepath)
```

步骤03 查看前两项数据。

下面的程序代码用来查看前两项泰坦尼克号的旅客数据。

```
In [5]: all_df[:2]
```

	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest
0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	S	2	NaN	St Louis, MO
1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.5500	C22 C26	S	11	NaN	Montreal, PQ / Chesterville, ON

以上各个字段说明见表 11-2。

表 11-2 字段说明

字段	字段说明	数据说明
survival	是否生存	0=否，1=是
pclass	舱等	1=头等舱，2=二等舱，3=三等舱

(续表)

字段	字段说明	数据说明
name	姓名	
sex	性别	female: 女性, male: 男性
age	年龄	
sibsp	手足或配偶也在船上数量	
parch	双亲或子女也在船上数量	
ticket	船票号码	
fare	旅客费用	
cabin	舱位号码	
embarked	登船港口	C=Cherbourg, Q=Queenstown, S=Southampton

以上字段中 survival (是否生存) 是 label 标签字段, 也就是我们要预测的目标, 其余都是特征字段。

步骤04 把需要的字段选取到 DataFrame 中。

以上字段中的 ticket (船票号码)、cabin (舱位号码), 我们认为与要预测的结果 survived (是否生存) 关联不大, 所以我们将其忽略, 只选择下列字段到 DataFrame 中。

```
In [6]: cols=['survived','name','pclass','sex','age','sibsp',
            'parch','fare','embarked']
        all_df=all_df[cols]
```

步骤05 选取字段后, 显示前两项数据。

```
In [7]: all_df[:2]
```

	survived	name	pclass	sex	age	sibsp	parch	fare	embarked
0	1	Allen, Miss. Elisabeth Walton	1	female	29.0000	0	0	211.3375	S
1	1	Allison, Master. Hudson Trevor	1	male	0.9167	1	2	151.5500	S

在以上字段中, 还有表 11-3 中的问题必须进行预处理, 后续才能够进行机器学习训练。

表 11-3 还需要处理的问题

字段	处理方式
name	姓名字段在预测阶段会使用, 但是训练时不需要, 必须先删除
age	有几项数据的 age 字段是 null 值, 所以必须将 null 值改为平均值
fare	有几项数据的 fare 字段是 null 值, 所以必须将 null 值改为平均值
sex	性别字段是文字, 我们必须转换为 0 与 1
embarked	分类特征字段有 3 个分类 C、Q、S, 必须使用 One-Hot Encoding 进行转换

11.3 使用 Pandas DataFrame 进行数据预处理

使用 Pandas DataFrame 进行数据预处理，后续才能够进行深度学习训练。

步骤01 将 name 字段删除。

下面的程序代码 all_df 使用 drop 方法删除 name 字段。

```
In [9]: df=all_df.drop(['name'], axis=1)
```

步骤02 找出含有 null 值的字段。

我们可以使用下列指令找出含有 null 值（无数据）的字段。

```
In [8]: all_df.isnull().sum()
Out[8]: survived    0
        name        0
        pclass     0
        sex        0
        age      263
        sibsp      0
        parch      0
        fare       1
        embarked   2
        dtype: int64
```

age 字段有263 项是null 值

fare 字段有1 项是null 值

embarked 字段有2 项是null 值

因为在后续进行深度学习训练时字段数据必须是数字，不能是 null 值，所以必须将 null 字段填上数值。至于要填上什么数值，最简单的方法是填上 0，但是 0 不符合实际状态，例如年龄应该不会是 0 岁，fare 运费应该也不会是 0，所以我们将 null 值替换为字段的平均值，这样比较符合实际情况。

步骤03 将 age 字段为 null 的数据替换成平均值。

下面的程序代码先使用 df['age'].mean()方法计算 age 字段的平均值 age_mean，然后使用 df['age'].fillna(age_mean)将 null 值替换成平均值。

```
In [10]: age_mean = df['age'].mean()
         df['age'] = df['age'].fillna(age_mean)
```

步骤04 将 fare 字段为 null 的数据替换成平均值。

下面的程序代码先使用 df['fare'].mean()方法计算 fare 字段的平均值 fare_mean，然后使用 df['fare'].fillna(fare_mean)将 null 值替换成平均值。

```
In [11]: fare_mean = df['fare'].mean()
         df['fare'] = df['fare'].fillna(fare_mean)
```


步骤05 转换性别字段为 0 与 1。

原本性别字段是文字，我们必须转换为 0 与 1，这样后续才能进行机器学习训练。以下程序代码使用 map 方法将'female'转换为 0，'male'转换为 1。

```
In [12]: df['sex'] = df['sex'].map({'female':0, 'male': 1}).astype(int)
```

步骤06 将 embarked 字段进行一位有效编码转换。

Pandas 提供了一个很方便的方法进行一位有效编码转换，使用 get_dummies()传入下列参数。

- **data:** 要转换的 DataFrame，在此输入 df。
- **columns:** 要转换的字段，在此输入["embarked"]。

```
In [14]: x_OneHot_df = pd.get_dummies(data=df, columns=["embarked" ])
```

步骤07 查看转换后的 DataFrame。

```
In [15]: x_OneHot_df[:2]
```

```
Out[15]:
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked_C	embarked_Q	embarked_S
0	1	1	0	29.0000	0	0	211.3375	0	0	1
1	1	1	1	0.9167	1	2	151.5500	0	0	1

将性别转换为0或1

原本embarked 字段转换为3 个字段

11.4 将 DataFrame 转换为 Array

因为后续要进行深度学习训练，所以必须先将 DataFrame 转换为 Array。

步骤01 DataFrame 转换为 Array。

```
In [16]: ndarray = x_OneHot_df.values
```

步骤02 查看 ndarray 的 shape。

可以使用下列指令来查看 array 的 shape。

```
In [17]: ndarray.shape
```

```
Out[17]: (1309, 10)
```

从以上执行结果可知，ndarray 共 1309 项 10 个字段。

步骤03 查看 ndarray 的前两项数据。

可以使用下列指令来查看 ndarray 的前两项数据。

第0个字段是label 第1个字段之后的字段是features

```
In [18]: ndarray[:2]
Out[18]: array([[ 1. ,  1. ,  0. , 29. ,  0. ,  0. ],
                [211.3375,  0. ,  0. ,  1. ],
                [ 1. ,  1. ,  1. , 0.9167,  1. ,  2. ],
                [151.55 ,  0. ,  0. ,  1. ]])
```

从以上执行结果可知，第 0 个字段是 label，第 1 个字段及之后的字段是 features。

步骤04 提取 features 与 label。

可以使用下列 Python slice 语句来提取 features 与 label。

```
In [19]: Label = ndarray[:,0]
         Features = ndarray[:,1:]
```

以上程序代码说明：ndarray 共有二维，第一维是项数，第二维是字段。

冒号提取所有项数 冒号提取所有项数

Label = ndarray[:,0] 第0个字段是label标签字段

Features = ndarray[:,1:] “1:”提取第1至最后字段，是特征字段

步骤05 查看前两项 label 标签字段。

```
In [23]: Label[:2]
Out[23]: array([ 1.,  1.])
```

步骤06 查看前两项 features 特征字段。

运费211元 年龄29岁

```
In [21]: Features[:2]
Out[21]: array([[ 1. ,  0. , 29. ,  0. ,  0. ],
                [211.3375,  0. ,  0. ,  1. ],
                [ 1. ,  1. , 0.9167,  1. ,  2. ],
                [151.55 ,  0. ,  0. ,  1. ]])
```

从以上执行结果可知，因为数值特征字段单位不同，例如年龄 29 岁、运费 211 元等，数字差异很大，没有一个共同的标准。这时就要使用标准化让所有数值都在 0 与 1 之间，使数值特征字段有共同的标准。进行标准化可以提高训练后模型的准确率。我们将在下一节介绍

如何进行标准化。

11.5 将 ndarray 特征字段进行标准化

我们将使用 sklearn 提供的 preprocessing 数据预处理模块进行标准化。

步骤01 导入 sklearn 的数据预处理模块。

```
In [24]: from sklearn import preprocessing
```

步骤02 建立 MinMaxScaler 标准化刻度 minmax_scale。

我们将使用 preprocessing.MinMaxScaler 来进行标准化，需输入参数 feature_range 设置标准化之后的范围在 0 和 1 之间。程序代码如下：

```
In [26]: minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))
```

步骤03 使用 minmax_scale.fit_transform 进行标准化。

然后使用 minmax_scale.fit_transform 传入参数 Features（特征字段）进行标准化。程序代码如下：

```
In [27]: scaledFeatures=minmax_scale.fit_transform(Features)
```

步骤04 查看标准化之后的特征字段前两项数据。

```
In [27]: scaledFeatures[:2]
Out[27]: array([[ 0.          ,  0.          ,  0.36116884,  0.          ,  0.
                ,  0.41250333,  0.          ,  0.          ,  1.          ],
                [ 0.          ,  1.          ,  0.00939458,  0.125       ,  0.222222
                ,  0.2958059 ,  0.          ,  0.          ,  1.          ]])
```

从以上执行结果可知，标准化之后的数字都介于 0 与 1 之间。

11.6 将数据分为训练数据与测试数据

因为在后面要进行深度学习模型的训练，所以必须将数据分为训练数据（用于训练模型）与测试数据（用于计算“训练完成模型”的准确率）。

步骤01 将数据以随机方式分为训练数据与测试数据。

```
In [30]: msk = numpy.random.rand(len(all_df)) < 0.8
        train_df = all_df[msk]
        test_df = all_df[~msk]
```

➤ 按照 8:2 的比例使用 numpy.random.rand 产生 msk

```
msk = numpy.random.rand(len(all_df)) < 0.8
```

➤ 产生训练数据，为全部数据的 80%

```
train_df = all_df[msk]
```

➤ 产生测试数据，为全部数据的 20%

```
test_df = all_df[~msk]
```

步骤02 显示训练数据与测试数据项数。

```
In [31]: print('total:',len(all_df),
              'train:',len(train_df),
              'test:',len(test_df))

total: 1309 train: 1053 test: 256
```

步骤03 创建 PreprocessData 函数进行数据的预处理。

我们将之前数据预处理的命令全部收集在 PreprocessData 函数中，方便后续使用。

```
In [32]: def PreprocessData(raw_df):
        df=raw_df.drop(['name'], axis=1)
        age_mean = df['age'].mean()
        df['age'] = df['age'].fillna(age_mean)
        fare_mean = df['fare'].mean()
        df['fare'] = df['fare'].fillna(fare_mean)
        df['sex']= df['sex'].map({'female':0, 'male': 1}).astype(int)
        x_OneHot_df = pd.get_dummies(data=df,columns=["embarked" ])

        ndarray = x_OneHot_df.values
        Features = ndarray[:,1:]
        Label = ndarray[:,0]

        minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))
        scaledFeatures=minmax_scale.fit_transform(Features)

        return scaledFeatures,Label
```

步骤04 对训练数据与测试数据进行预处理。

```
In [33]: train_Features,train_Label=PreprocessData(train_df)
        test_Features,test_Label=PreprocessData(test_df)
```

数据预处理后的结果如下：

- train_Features (训练数据的特征字段)，train_Label (训练数据的标签字段)。
- test_Features (测试数据的特征字段)，test_Label (测试数据的标签字段)。

步骤05 查看数据预处理后训练数据的特征字段。

数据预处理之后，查看 `train_Features`（训练数据的特征字段）的前两项数据。

```
In [34]: train_Features[:2]
Out[34]: array([[ 0.          ,  0.          ,  0.36116884,  0.          ,  0.
                ,  0.41250333,  0.          ,  0.          ,  1.          ],
                [ 0.          ,  1.          ,  0.00939458,  0.125       ,  0.222
                22222,  0.2958059 ,  0.          ,  0.          ,  1.          ]])
```

步骤06 查看数据预处理后训练数据的标签字段。

数据预处理之后，查看 `train_Label`（训练数据的标签字段）的前两项数据。

```
In [36]: train_Label[:2]
Out[36]: array([ 1.,  1.])
```

11.7 结论

在本章中，我们介绍了下载并且读取泰坦尼克号的旅客数据集，并介绍了泰坦尼克号数据集的特色，最后完成数据的预处理。在下一章，我们就可以使用 Keras 建立多层感知器模型，训练模型并进行预测。

第12章

Keras多层感知器预测泰坦尼克号上 旅客的生存概率

在本章中，我们将建立多层感知器模型，训练模型、评估模型的准确率，然后使用训练完成的模型来预测泰坦尼克号上旅客生存的概率，并预测《泰坦尼克号》电影中男女主角生存的概率，找出泰坦尼克号上其他旅客的感人故事。

我们将建立多层感知器模型，如图 12-1 所示。

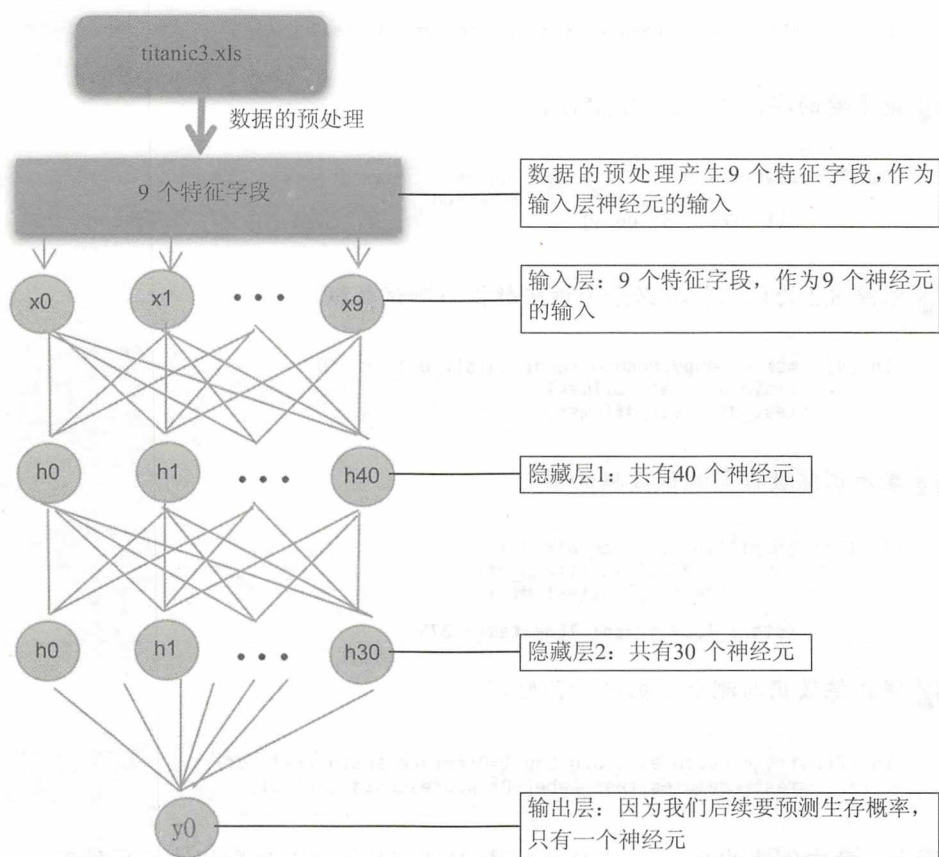


图 12-1

本章完整的程序代码可参考范例程序 Keras_Taianic_MLP.ipynb。范例程序下载与安装可参考本书附录 A。

12.1 数据预处理

以下数据预处理的详细说明可参考 11 章的内容。

步骤01 导入所需模块。

```
In [1]: import urllib.request
import os
import tarfile
```

步骤02 读取泰坦尼克号的数据集文件。

```
In [2]: all_df = pd.read_excel("data/titanic3.xls")
```

步骤03 把需要的字段选取到 DataFrame 中。

```
In [3]: cols=['survived','name','pclass','sex','age','sibsp',  
            'parch','fare','embarked']  
all_df=all_df[cols]
```

步骤04 依照 8:2 的比例将数据分为训练数据与测试数据。

```
In [4]: msk = numpy.random.rand(len(all_df)) < 0.8  
train_df = all_df[msk]  
test_df = all_df[~msk]
```

步骤05 显示训练数据与测试数据的项数。

```
In [5]: print('total:',len(all_df),  
            'train:',len(train_df),  
            'test:',len(test_df))  
  
total: 1309 train: 1034 test: 275
```

步骤06 将训练数据与测试数据进行预处理。

```
In [7]: train_Features,train_Label=PreprocessData(train_df)  
test_Features,test_Label=PreprocessData(test_df)
```

使用上一章中创建的 PreprocessData 函数对训练数据与测试数据进行预处理：

- train_Features (训练数据的特征字段)，train_Label (训练数据的标签字段)。
- test_Features (测试数据的特征字段)，test_Label (测试数据的标签字段)。

12.2 建立模型

我们将使用下面的程序代码建立多层感知器模型：输入层（9 个神经元）、隐藏层 1（40 个神经元）、隐藏层 2（30 个神经元）、输出层（1 个神经元），如图 12-2 所示。

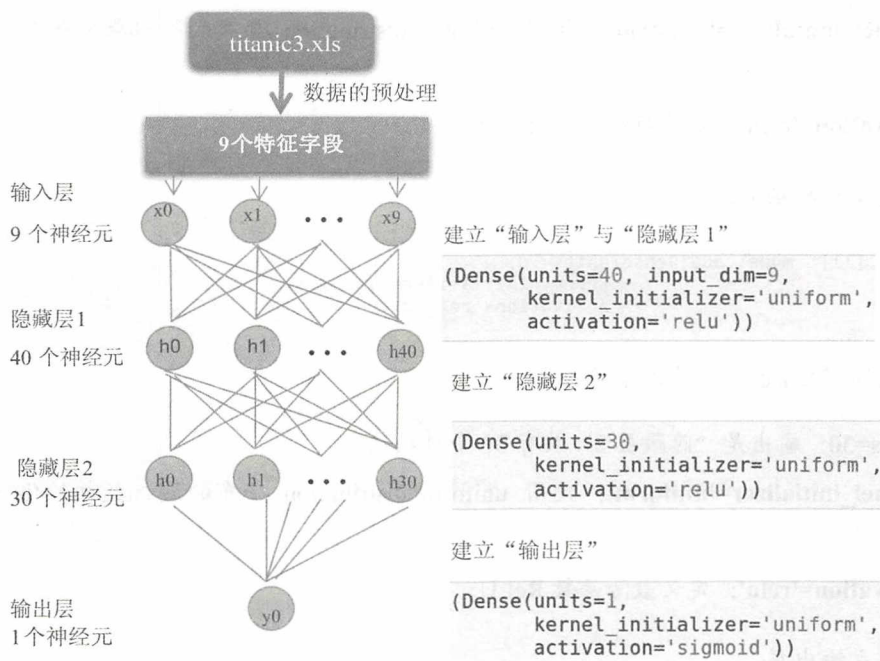


图 12-2

步骤01 导入所需模块。

```
In [8]: from keras.models import Sequential
from keras.layers import Dense, Dropout
Using TensorFlow backend.
```

步骤02 建立 keras Sequential 模型。

下面的程序代码建立一个线性堆叠模型，后续只需要将各个神经网络层加入模型即可。

```
In [9]: model = Sequential()
```

步骤03 建立输入层与隐藏层 1。

下面的程序代码使用 `model.add` 方法加入 Dense 神经网络层。Dense 神经网络层的特色是：所有的上一层与下一层的神经元都完全连接。

```
In [10]: model.add(Dense(units=40, input_dim=9,
kernel_initializer='uniform',
activation='relu'))
```

model 加入 Dense 层，需输入下列参数。

- **units=40**: 输出是“隐藏层 1”共有 40 个神经元。
- **input_dim=9**: 输入层有 9 个神经元的输入，因为数据预处理后有 9 个特征字段。

- **kernel_initializer='uniform'**: 使用 uniform distribution 分布的随机数初始化 weight 与 bias。
- **activation='relu'**: 定义激活函数 ReLU。

步骤04 建立隐藏层 2。

```
In [11]: model.add(Dense(units=30,  
                        kernel_initializer='uniform',  
                        activation='relu'))
```

model 加入 Dense 层需输入下列参数。

- **units=30**: 输出是“隐藏层 2”共有 30 个神经元。
- **kernel_initializer='uniform'**: 使用 uniform distribution 分布的随机数初始化 weight 与 bias。
- **activation='relu'**: 定义激活函数 ReLU。

步骤05 建立输出层。

```
In [12]: model.add(Dense(units=1,  
                        kernel_initializer='uniform',  
                        activation='sigmoid'))
```

model 加入 Dense 层需输入下列参数。

- **units=1**: “输出层”共有 1 个神经元。
- **kernel_initializer='uniform'**: 使用 uniform distribution 分布初始化 weight 与 bias。
- **activation='relu'**: 定义激活函数 ReLU。

12.3 开始训练

当我们建立好深度学习模型后，就可以使用反向传播算法进行训练。有关使用反向传播算法进行训练的说明可参考第 2 章。

1. 定义训练方式

在训练模型之前，我们必须使用 compile 方法对训练模型进行设置，指令如下：

```
In [13]: model.compile(loss='binary_crossentropy',  
                      optimizer='adam', metrics=['accuracy'])
```



compile 方法需输入 3 个参数: loss、optimizer 和 metrics (对这 3 个参数的解释说明可参考 7.4 节)。

2. 开始训练

执行训练的程序代码如下:

```
In [14]: train_history =model.fit(x=train_Features,
                                   y=train_Label,
                                   validation_split=0.1,
                                   epochs=30,
                                   batch_size=30,verbose=2)
```

以上程序代码说明如下:

使用 model.fit 进行训练, 训练过程会存储在 train_history 变量中, 这个训练需输入下列参数。

(1) 输入训练数据的参数

- `x=train_Features`, features 共 9 个特征字段。
- `y=train_Label`, label 标签字段 (是否生存? 是: 1, 否: 0)。

(2) 设置训练与验证数据的比例

- 设置参数 `validation_split=0.1`。

训练之前 Keras 会自动将数据分成: 90%作为训练数据, 10%作为验证数据。因为全部是 1034 项, 所以分成: $1034 \times 0.9 = 930$ 作为训练数据, $1034 \times 0.1 = 104$ 作为验证数据。

(3) 设置训练周期次数与每一批次的项数

- `epochs=30`: 执行 30 个训练周期。
- `batch_size=30`: 每一批次 30 项数据。

共执行了 30 个训练周期, 说明如下:

- 每一个训练周期, 使用 930 项训练数据进行训练, 分为每一批次 30 项, 所以大约分为 31 个批次 ($930/30=31$) 进行训练。
- 这个训练周期完成后, 计算此次训练周期后的准确率与误差。

(4) 设置显示训练过程

- `verbose=2`: 显示训练过程。

➤ 以上程序代码执行后结果如图 12-3 所示。

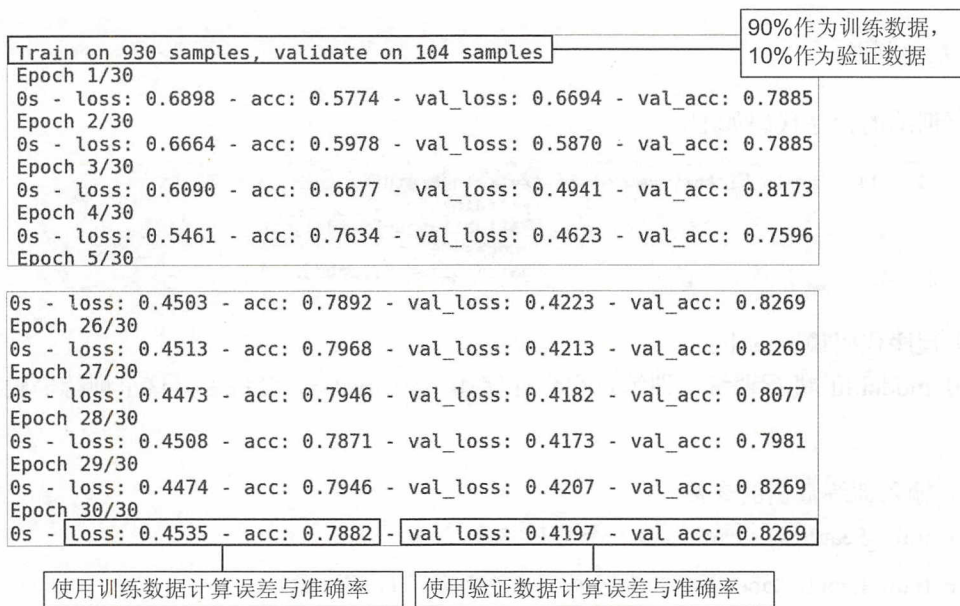
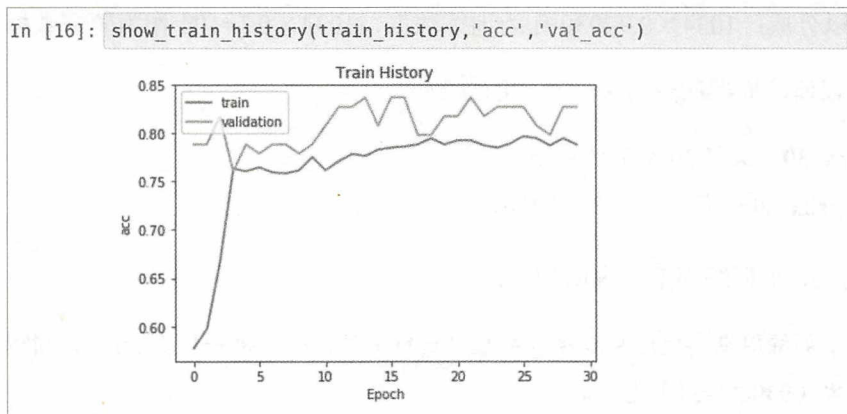


图 12-3

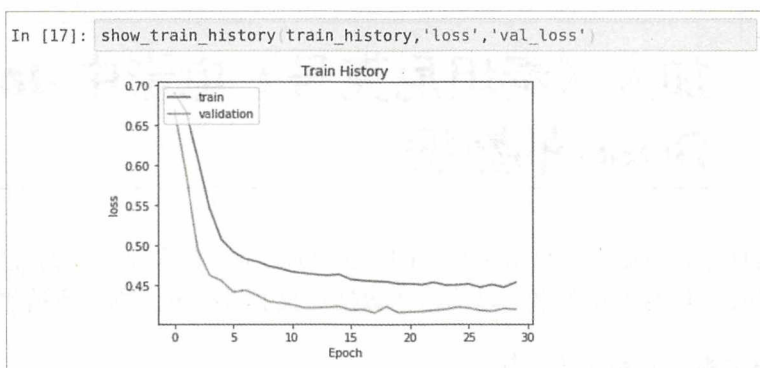
3. 画出准确率的执行结果

使用下面的程序代码画出准确率的执行结果。



在以上执行结果的屏幕显示界面中，“acc 训练的准确率”是深色的，“val_acc 验证的准确率”是浅色的，总共执行了 10 个训练周期，我们可以发现：无论是训练还是验证，准确率都越来越高。

4. 画出误差的执行结果



在以上执行结果的屏幕显示界面中，我们可以看到共执行了 30 个训练周期，“loss 训练的误差”是深色的，“val_loss 验证的误差”是浅色的，我们还可以发现，无论是训练还是验证，误差都越来越低。

12.4 评估模型准确率

之前我们已经训练完模型，现在要使用 test 测试数据集来评估模型的准确率。

1. 评估模型的准确率

下面的程序代码用来评估模型的准确率。

```
In [18]: scores = model.evaluate(x=test_Features,
                                y=test_Label)
32/275 [==>.....] - ETA: 0s
```

以上程序代码说明见表 12-1。

表 12-1 程序代码说明

程序代码	说明
scores=model.evaluate	使用 model.evaluate 评估模型的准确率，评估后的准确率会存储在 scores 中
x= test_Features	测试数据的 features 共有 9 个特征字段
y= test_Label	测试数据的 label 标签字段（是否存活？是：1，否：0）

2. 查看评估的准确率

```
In [19]: scores[1]
Out[19]: 0.80363636406985195
```

从以上执行结果可知准确率是 0.80。

12.5 加入《泰坦尼克号》电影中 Jack 与 Rose 的数据

在《泰坦尼克号》电影中，男女主角 Jack 与 Rose 是虚构人物，我们希望能用所训练完成的模型预测男女主角的生存概率。以下这些数据是我们根据电影剧情所猜想的：

- Jack 是 3 等舱，Rose 是头等舱。
- Jack 是男性，Rose 是女性。
- Jack 的票价是 5，Rose 的票价是 100。
- Jack 的年龄是 23，Rose 的年龄是 20。

步骤01 建立 Jack 与 Rose 的数据。

使用 pd.Series 建立 Jack 与 Rose 的数据如下：

```
In [20]: Jack = pd.Series([0, 'Jack', 3, 'male', 23, 1, 0, 5.0000, 'S'])
         Rose = pd.Series([1, 'Rose', 1, 'female', 20, 1, 0, 100.0000, 'S'])
```

步骤02 创建 Jack 与 Rose 的 DataFrame。

使用 pd.DataFrame 创建 Pandas DataFrame JR_df，加入 Jack 与 Rose 的数据。

```
In [21]: JR_df = pd.DataFrame([list(Jack), list(Rose)],
                               columns=['survived', 'name', 'pclass', 'sex',
                                         'age', 'sibsp', 'parch', 'fare', 'embarked'])
```

步骤03 将 JR_df 加入 all_df。

因为我们后续要使用 all_df 进行预测，所以我们将 JR_df 加入 all_df。

```
In [22]: all_df=pd.concat([all_df, JR_df])
```

步骤04 查看 all_df 最后两项数据。

将 JR_df 加入 all_df 后，最后两项数据就是 Jack 与 Rose 的数据。

```
In [23]: all_df[-2:]
Out[23]:
```

	survived	name	pclass	sex	age	sibsp	parch	fare	embarked
0	0	Jack	3	male	23.0	1	0	5.0	S
1	1	Rose	1	female	20.0	1	0	100.0	S

12.6 进行预测

在前面的步骤中我们建立了模型，并且完成了模型的训练，准确率达到还可以接受的 0.80，接下来我们将使用此模型进行预测。

步骤01 执行数据预处理。

因为 Jack 与 Rose 的数据是后来才加入的，所以必须再次执行数据预处理。

```
In [24]: all_Features,Label=PreprocessData(all_df)
```

步骤02 执行预测。

使用 `model.predict` 传入参数 `all_Features`（特征字段）执行预测，返回预测结果 `all_probability`。

```
In [25]: all_probability=model.predict(all_Features)
```

步骤03 预测结果。

使用下列指令来查看预测结果 `all_probability` 的前 10 项数据。

```
In [26]: all_probability[:10]
Out[26]: array([[ 0.97593725],
 [ 0.59155542],
 [ 0.97304821],
 [ 0.38359511],
 [ 0.97123849],
 [ 0.27242622],
 [ 0.9457652 ],
 [ 0.32310808],
 [ 0.9449923 ],
 [ 0.30549577]], dtype=float32)
```

第1位旅客的生存概率

以上预测结果其实就是每一位旅客的生存概率。

步骤04 将 `all_df` 与 `all_probability` 整合。

接下来，我们将 `all_df`（姓名与所有特征字段）与 `all_probability`（预测结果）整合产生 `pd DataFrame`。

```
In [27]: pd=all_df
pd.insert(len(all_df.columns),
        'probability',all_probability)
```


步骤05 查看预测《泰坦尼克号》电影中 Jack 与 Rose 生存概率的结果。

我们可以使用 `pd[-2:]` 选取 DataFrame 最后两项数据，那就是 Jack 与 Rose 生存概率的结果。

In [29]:

pd[-2:]

Out[29]:

	survived	name	pclass	sex	age	sibsp	parch	fare	embarked	probability
0	0	Jack	3	male	23.0	1	0	5.0	S	0.146854
1	1	Rose	1	female	20.0	1	0	100.0	S	0.969164

从以上执行结果可知，Jack 的生存概率只有 0.14，Rose 的生存概率高达 0.96，符合电影最后的结局。

12.7 找出泰坦尼克号背后的感人故事

数据科学家在处理数据时，常常只会看到冷冰冰的数据，而忘记这些数据背后的故事，每一项数据都曾经是活生生的人，他们都有父母家人，有很多感人的故事。

1. 查看生存概率高，却没有存活的旅客

我们也许会好奇，根据模型，哪些旅客预测生存概率高，可是却没有存活？Pandas 提供了很方便的功能，可以让我们按照条件查询所需要的数据。

使用下列 pandas 的语句来查询生存概率大于 90%但是没有存活的数据，结果如图 12-4 所示。

- `pd['probability'] > 0.9`: 生存概率大于 90%。
- `pd['survived'] == 0`: 没有存活。

```
In [31]: pd[(pd['survived']==0) & (pd['probability']>0.9) ]
```

	survived	name	pclass	sex	age	sibsp	parch	fare	embarked	probability
2	0	Allison, Miss. Helen Loraine	1	female	2.0	1	2	151.5500	S	0.977254
4	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	1	female	25.0	1	2	151.5500	S	0.974557
105	0	Evans, Miss. Edith Corse	1	female	36.0	0	0	31.6792	C	0.979168
169	0	Isham, Miss. Ann Elizabeth	1	female	50.0	0	0	28.7125	C	0.975072
286	0	Straus, Mrs. Isidor (Rosalie Ida Blun)	1	female	63.0	1	0	221.7792	S	0.949332

图 12-4

图 12-4 的数据中，Allison, Miss. Helen Loraine 与 Allison, Mrs. Hudson J C 都是 Allison 家族的人，他们依照我们的模型生存概率高，可是却没有存活，到底发生了什么事呢？



2. Allison 家族的故事

显示前 5 项预测结果（见图 12-5）：

In [28]: pd[:5]

	survived	name	pclass	sex	age	sibsp	parch	fare	embarked	probability
0	1	Allen, Miss. Elisabeth Walton	1	female	29.0000	0	0	211.3375	S	0.979868
1	1	Allison, Master. Hudson Trevor	1	male	0.9167	1	2	151.5500	S	0.589231
2	0	Allison, Miss. Helen Loraine	1	female	2.0000	1	2	151.5500	S	0.977254
3	0	Allison, Mr. Hudson Joshua Creighton	1	male	30.0000	1	2	151.5500	S	0.356341
4	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	1	female	25.0000	1	2	151.5500	S	0.974557

Allison 家族成员

图 12-5

真实沉船中 Allison 家族的故事是，Allison 一家人共有 4 位成员，即爸爸（30 岁）、妈妈（25 岁）、一个两岁的女儿 Loraine 以及一个不满一岁的婴儿 Trevor。他们全家加上一名护士（Alice Cleaver）乘坐邮轮返回加拿大蒙特利尔。

因为搭乘救生艇女士优先，原本妈妈可以带着女儿与小婴儿上救生艇，但是因为找不到婴儿 Trevor，所以坚持不愿意上救生艇，而在船上到处寻找婴儿，最后全家一起在船上沉没。

然而命运捉弄人的是，原来小婴儿 Trevor 早就被护士（Alice Cleaver）带上救生艇，但是没有告知 Allison 家人，导致全家人都找不到婴儿。最后 Allison 全家只有不满一岁的小婴儿 Trevor 存活，但是失去了所有亲人。还好后来小婴儿 Trevor 回到加拿大，由他的叔叔婶婶抚养长大。

3. 爱狗女士的故事

查询生存概率大于 90%却没有存活的数据，我们可以看到 Ann Elizabeth Isham，如图 12-6 所示。

	survived	name	pclass	sex	age	sibsp	parch	fare	embarked	probability
2	0	Allison, Miss. Helen Loraine	1	female	2.0	1	2	151.5500	S	0.977254
4	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	1	female	25.0	1	2	151.5500	S	0.974557
105	0	Evans, Miss. Edith Corse	1	female	36.0	0	0	31.6792	C	0.979168
169	0	Isham, Miss. Ann Elizabeth	1	female	50.0	0	0	28.7125	C	0.975072
286	0	Straus, Mrs. Isidor (Rosalie Ida Blun)	1	female	63.0	1	0	221.7792	S	0.949332

Ann Elizabeth Isham

图 12-6

一位 50 岁的女乘客 Ann Elizabeth Isham，她住在巴黎，要搭船返回美国，带着她的爱狗搭乘泰坦尼克号，于是将自己心爱的大丹狗安置在船上的狗舍，并且每日探望。发生船难时，Ann 问自己的狗是否可以上救生艇，但是被拒绝。然而 Ann 实在无法抛弃心爱的狗独自

逃生，对 Ann 而言，狗是她的家人，于是舍弃生还机会，宁愿与爱狗共存亡。

4. 施特劳斯夫妇的故事

查询生存概率大于 90%却没有存活的数据，我们还可以看到 Struss Isidor，如图 12-7 所示。

	survived	name	pclass	sex	age	sibsp	parch	fare	embarked	probability
2	0	Allison, Miss. Heien Loraine	1	female	2.0	1	2	151.5500	S	0.977254
4	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	1	female	25.0	1	2	151.5500	S	0.974557
105	0	Evans, Miss. Edith Corse	1	female	36.0	0	0	31.6792	C	0.979168
169	0	Isham, Miss. Ann Elizabeth	1	female	50.0	0	0	28.7125	C	0.975072
286	0	Straus, Mrs. Isidor (Rosalie Ida Blun)	1	female	63.0	1	0	221.7792	S	0.949332

Straus, Mrs. Isidor(Rosalie Ida Blun)

图 12-7

伊思德·施特劳斯 (Isidor Straus) 是美国知名百货梅西百货 (Macy's) 的共同创办人，与他的太太艾达·施特劳斯 (Ida Straus) 登上泰坦尼克号，准备从欧洲回到美国的家。然而发生船难时，得知轮船即将沉没，埃达本来有机会逃生，但是不愿意离开丈夫，拒绝独自乘坐救生船逃生。于是埃达将自己的皮大衣送给了女仆埃伦 (Ellen)，让埃伦登上了救生船。据救生艇上的目击者说，当时记得艾达对丈夫说：“我们已经一起生活这么多年了，无论你在哪，我就去哪 (Where you go, I go)”。

12.8 结论

在本章中，我们建立了多层感知器模型，经过数据预处理、训练模型，使用训练完成的模型来预测泰坦尼克号上旅客的生存概率，并且找出泰坦尼克号背后的感人故事。在下一章我们将介绍 IMDb 网络电影数据库与自然语言处理。

第13章

IMDb网络电影数据集与自然语言处理

情感分析 (sentiment analysis) 又称为意见挖掘 (opinion mining), 是使用“自然语言处理”、文字分析等方法找出作者某些话题上的态度、情感、评价或情绪。情感分析的商业价值在于, 可提早得知顾客对公司或产品的观感, 以调整销售策略的方向。第13、14章我们将介绍 IMDb 网络电影数据集, 使用词嵌入 (Word Embedding) 自然语言处理的方法进行预处理, 并且建立各种深度学习模型, 进行情感分析。

IMDb 网络电影数据库（Internet Movie Database）是一个与电影相关的在线数据库。IMDb 始于 1990 年，自 1998 年起成为亚马逊旗下的网站，至今已经累积了大量的电影信息。IMDb 收录了共 400 多万部电影作品数据。IMDb 的网址为 <http://www.imdb.com/>。

IMDb 数据集共有 50 000 项“影评文字”，分为训练数据与测试数据各 25 000 项，每一项“影评文字”都被标记为“正面评价”或“负面评价”。

我们希望能建立一个模型，经过大量“影评文字”训练后，此模型可以用于预测“影评文字”是“正面评价”或“负面评价”。例如图 13-1 所示的深度学习模型识别 IMDb “影评文字”，可分为训练与预测。

训练（Training）

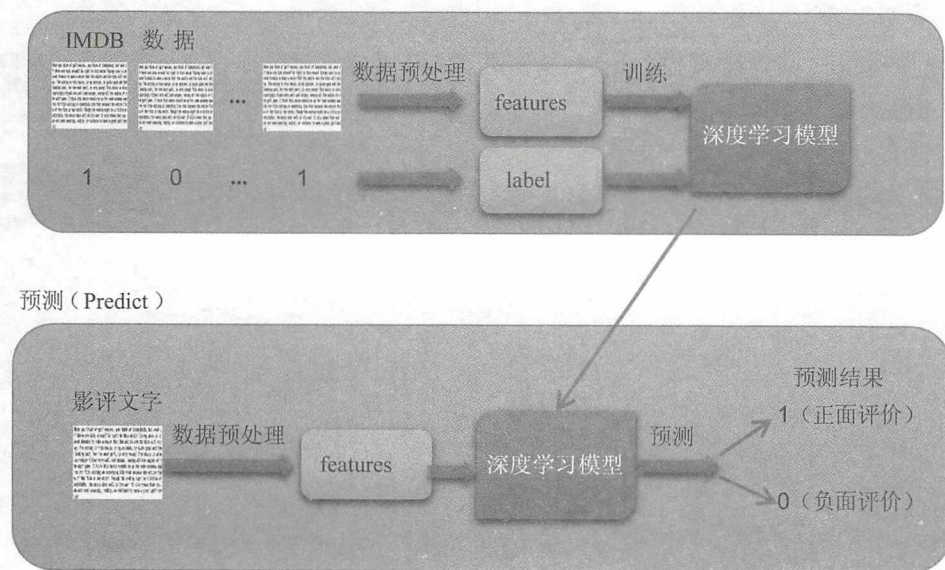


图 13-1

图 13-1 说明如下：

➤ 训练

IMDb 数据集的训练数据共 25 000 项，经过数据预处理后会产生 features（特征值）与 label（1：正面评价，0：负面评价），然后对深度学习模型进行训练，训练完成的模型就可以在下一阶段预测时使用。

➤ 预测

输入“影评文字”，预处理后会产生 features（特征值），可以使用训练完成的多层感知器模型进行预测，最后产生预测结果（“正面评价”或“负面评价”）。

本章完整的程序代码参考 `keras_Imdb_Introduce.ipynb`。范例下载与安装参考附录 A。

13.1 Keras 自然语言处理介绍

Keras 自然语言处理 IMDB 影评文字步骤如图 13-2 所示。

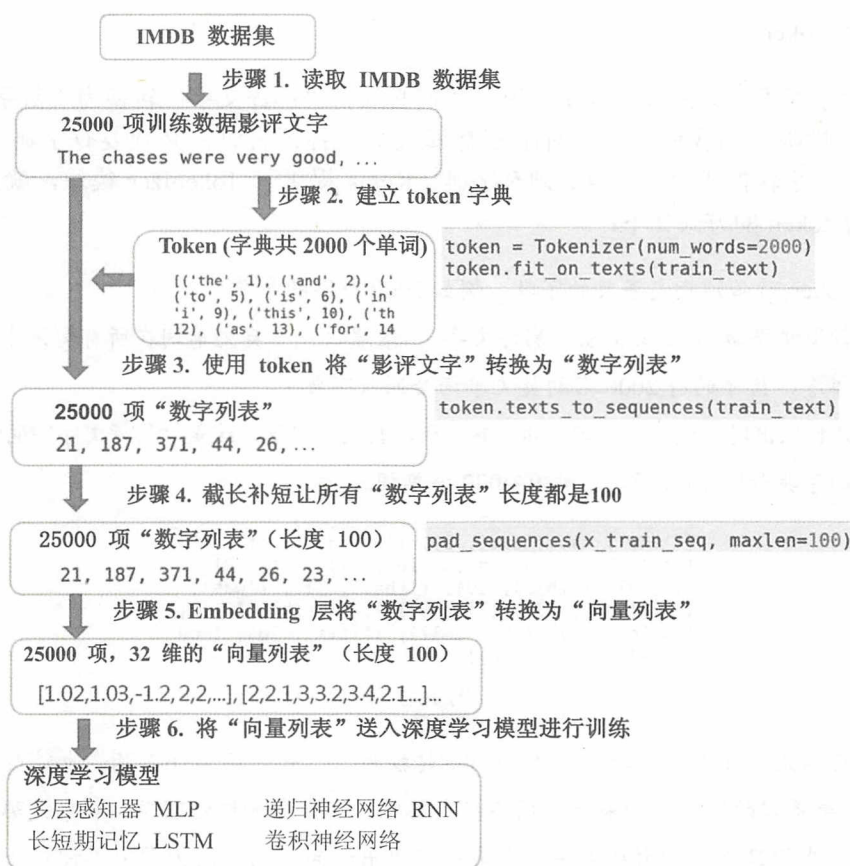


图 13-2

图 13-2 的步骤说明如下：

步骤01 读取 IMDB 数据集。

IMDB 数据集分为训练数据与测试数据，说明见表 13-1。

表 13-1 训练数据与测试数据说明

训练数据	train_text (文字)	0 到 12499 项：正面评价影评文字 12500 到 24999 项：负面评价影评文字
	y_train (标签)	0 到 12499 项：正面评价，全部是 1 12500 到 24999 项：负面评价，全部是 0

(续表)

测试数据	test_text (文字)	0 到 12499 项: 正面评价影评文字 12500 到 24999 项: 负面评价影评文字
	y_test (标签)	0 到 12499 项: 正面评价, 全部是 1 12500 至 24999 项: 负面评价, 全部是 0

步骤02 建立 token。

因为深度学习模型只能接受数字, 所以我们将“影评文字”转换为“数字列表”。

要如何转换呢? 当我们要将一种语言翻译成另一种语言时, 必须要有字典。相同的道理, 我们要将文字转换成数字, 也必须有字典。Keras 提供了 Tokenizer 模块, 就是类似字典的功能。建立 token 的方式如下:

- 建立 token 时必须指定字典的字数, 例如 2000 个字的字典。
- 然后读取训练数据 25000 项“影评文字”, 依照每一个英文单词在所有影评中出现的次数进行排序, 排序的前 2000 名的英文单词会列入字典中。
- 因为是按照出现次数排序所建立的字典, 所以我们可以说, 这是“影评文字”的常用字典。
- 建立的字典如图 13-3 所示, 共有 2000 个单词。

```
[('the', 1), ('and', 2), ('a', 3), ('of', 4),
('to', 5), ('is', 6), ('in', 7), ('it', 8), ('i', 9),
('this', 10), ('that', 11), ('was', 12), ('as', 13),
('for', 14), ('with', 15), ('movie', 16), ('but', 17),
('film', 18), ('on', 19), ('not', 20)]
```

图 13-3

- 我们可以用此字典进行转换, 例如 'the'转换为 1、'is'转换为 6。读者也许会好奇, 只有 2000 个单词的字典, 如果有单词不在字典中, 那么会如何处理呢? 答案是那个单词就不转换, 我们只在乎“影评文字”在常用字典出现的单词, 因为常用单词对于我们要预测的目标影响比较大, 不常用单词对于我们后续的预测影响比较小。

步骤03 使用 token 将“影评文字”转换为“数字列表”。

建立 token 字典后, 我们就可以使用 token 将“影评文字”转换为“数字列表”。

例如, 将图 13-4 中的“影评文字”转换为“数字列表”。

The chases were very good

↓ 转换

21, 187, 371, 44, 26

图 13-4

我们会将 25 000 项“影评文字”训练数据转换为 25 000 项的“数字列表”。

步骤04 截长补短让所有“数字列表”长度为 100。

因为每一则“影评文字”的字数都不固定，例如有些可能有 170 个字，有些有 80 个字。转换成“数字列表”字数也不固定。因为后续要将“数字列表”转为“向量列表”，并送入深度学习模型进行训练，所以长度必须固定。如何让所有“数字列表”长度都固定呢？其实方法很简单，就是截长补短。

例如，我们要将“数字列表”的长度都设置为 100。

- 如果数字列表的长度是 59，就在前面补上 41 个“0”，这样就变成长度为 100 的“数字列表”，如图 13-5 所示。
- 如果数字列表的长度是 126，就将前面的 26 个数字截去，这样就变成长度为 100 的“数字列表”。

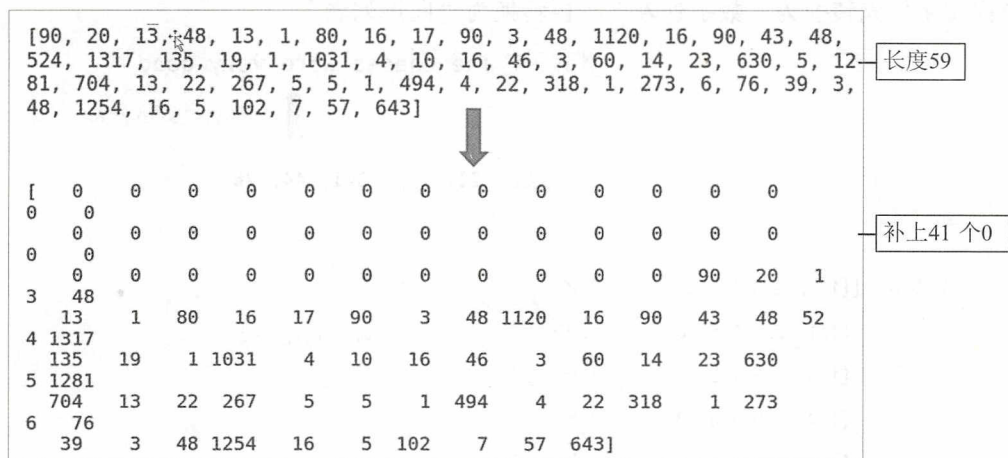


图 13-5

步骤05 使用嵌入层将“数字列表”转换为“向量列表”。

词嵌入是一种自然语言处理技术，其原理是将文字映射成多维几何空间的向量。语义类似的文字向量在多维的几何空间的距离也比较相近。前面我们将“影评文字”转换为数字，可是数字在语义上无任何关联。为了能让每一个文字有关联性，必须转换为向量。

➤ 文字转换为数字，数字在语义上无任何关联

pleasure ➡ 38	like ➡ 10	attraction ➡ 313
dislike ➡ 21	hate ➡ 28	disgust ➡ 31

➤ 文字转换为向量，语义类似的文字，在向量空间也会比较接近（见图 13-6）

pleasure ➡ 38 ➡ (1.2, 2.3, 3.2)	dislike ➡ 21 ➡ (-1.21, 2.7, 3.2)
like ➡ 10 ➡ (1.25, 2.33, 3.4)	hate ➡ 28 ➡ (-1.5, 2.63, 3.22)
attraction ➡ 313 ➡ (1.25, 1.33, 3.3)	disgust ➡ 31 ➡ (-1.65, 1.83, 3.11)

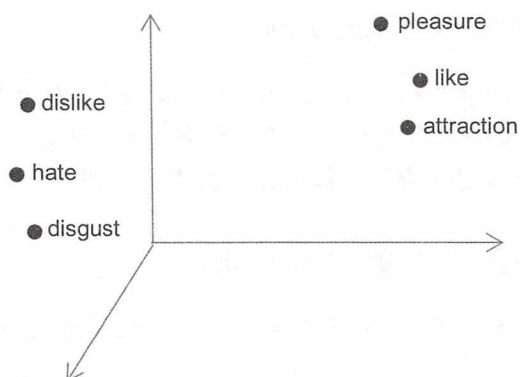


图 13-6

Keras 提供了嵌入层可以用于将“数字列表”转换为“向量列表”。例如，将图 13-7 中的“影评文字”先转换为“数字列表”，再转换为“向量列表”

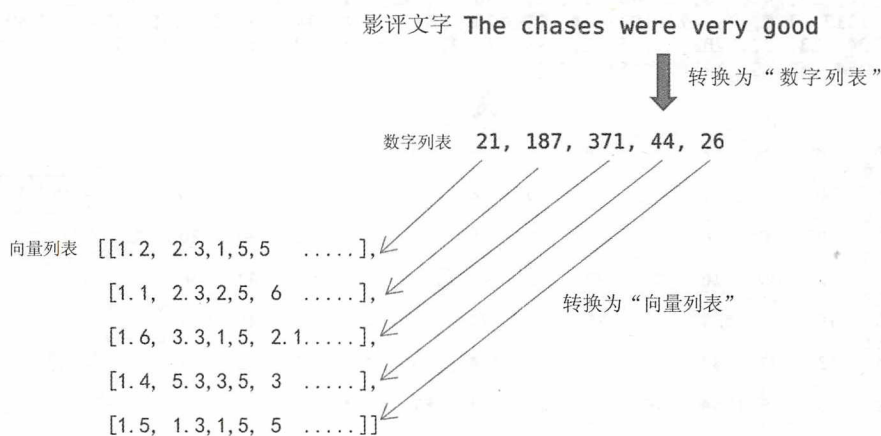


图 13-7

步骤06 将“向量列表”送入“深度学习模型”进行训练。

在前面的步骤中，我们将“影评文字”先转换为“数字列表”，再转换为“向量列表”后，就可以将“向量列表”送入深度学习模型进行训练，下一章我们将介绍使用下列深度学习模型进行训练。

深度学习模型

多层感知器 MLP
长短期记忆 LSTM

递归神经网络
RNN

“深度学习模型”训练完成后，就可以进行预测了，如图 13-8 所示。

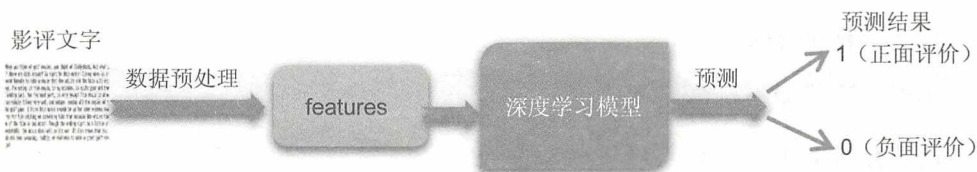


图 13-8

以上步骤说明如下。

- 步骤 1~4: 文字的预处理, 将在本章中介绍。
- 步骤 5 和 6: 建立嵌入层, 并且使用深度学习模型进行训练与预测, 将在下一章介绍。

13.2 下载 IMDb 数据集

可以在下列网址下载 IMDb 数据集:

<http://ai.stanford.edu/~amaas/data/sentiment>

1. 导入所需模块

```
In [1]: import urllib.request
import os
import tarfile
```

以上程序代码说明见表 13-2。

表 13-2 程序代码说明

程序代码	说明
<code>import urllib.request</code>	导入 <code>urllib</code> 模块, 将用于下载文件
<code>import os</code>	导入 <code>os</code> 模块, 用于确认文件是否存在
<code>import tarfile</code>	导入 <code>tarfile</code> 模块, 用于解压缩文件

2. 下载 IMDb 数据集

使用下列程序代码下载 IMDb 数据集。

```
In [2]: url="http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"
filepath="data/aclImdb_v1.tar.gz"
if not os.path.isfile(filepath):
    result=urllib.request.urlretrieve(url,filepath)
    print('downloaded:',result)

downloaded: ('data/aclImdb_v1.tar.gz', <http.client.HTTPMessage object at 0x7ff734109c18>)
```

从以上执行结果可知, 下载了 `aclImdb_v1.tar.gz`, 并存储在程序执行目录下的 `data` 目录下。

➤ 设置下载的网址

```
url="http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"
```

➤ 设置存储文件的路径

```
filepath="data/aclImdb_v1.tar.gz"
```

➤ 判断文件不存在就会下载文件

```
if not os.path.isfile(filepath): result=urllib.request.urlretrieve(url,
filepath) print('downloaded:',result)
```

上面的程序代码在判断文件不存在时，就会使用 `urllib.request.urlretrieve` 下载文件。输入参数：`url`（下载的网址）与 `filepath`（存储文件的路径）。

3. 解压缩下载的文件

```
In [3]: if not os.path.exists("data/aclImdb"):
        tfile = tarfile.open("data/aclImdb_v1.tar.gz", 'r:gz')
        result=tfile.extractall('data/')
```

以上程序代码说明如下：

➤ 判断解压缩目录是否存在

```
if not os.path.exists("data/aclImdb"):
```

➤ 打开压缩文件

```
tfile = tarfile.open("data/aclImdb_v1.tar.gz", 'r:gz')
```

➤ 解压缩文件到 data 目录中

```
result=tfile.extractall('data/')
```

解压缩完成后，产生了 `aclImdb` 目录，后文会介绍这个目录的内容。

4. 查看已下载的文件及解压缩目录

查看下载的数据文件，根据使用的环境是 Windows 或 Linux Ubuntu 而稍有不同，说明如下：

➤ 在 Windows 下查看已下载的数据文件

可以使用文件资源管理器来查看程序执行目录下的 `data` 目录，例如程序执行目录是 `C:\pythonwork\keras`，就可以在 `C:\pythonwork\keras\data` 中看到已下载的 `aclImdb_v1.tar` 文件与解压缩目录 `aclImdb`，如图 13-9 所示。

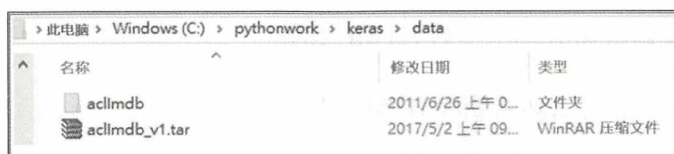


图 13-9

➤ 在 Linux Ubuntu 下查看已下载的数据文件

在“终端”程序输入下列命令，先切换到程序执行目录，再查看目录。

```
cd ~/pywork/keras/data ll
```

执行后屏幕显示界面如图 13-10 所示，我们可以看到已下载的 aclImdb_v1.tar 文件与解压缩目录 aclImdb。

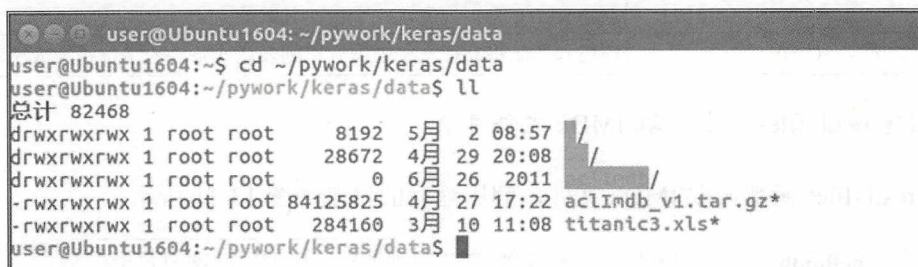


图 13-10

13.3 读取 IMDb 数据

IMDb 文件下载并解压缩后，共有 50 000 项文本文件，我们将以下列步骤来读取，并且把它们分为训练数据与测试数据。

1. 导入所需模块

```
In [1]: from keras.preprocessing import sequence
        from keras.preprocessing.text import Tokenizer

        Using TensorFlow backend.
```

程序代码说明见表 13-3。

表 13-3 程序代码说明

程序代码	说明
from keras.preprocessing.text import Tokenizer	导入 Tokenizer 模块，将用于建立字典
from keras.preprocessing import sequence	导入 sequence 模块，将用于截长补短让所有“数字列表”长度为 100



2. 创建 rm_tag 函数删除文字中的 HTML 标签

下列程序代码使用正则表达式删除 HTML 的标签。

```
In [4]: import re
def rm_tags(text):
    re_tag = re.compile(r'<[^>]+>')
    return re_tag.sub('', text)
```

程序代码说明见表 13-4。

表 13-4 程序代码说明

程序代码	说明
import re	导入 Regular Expression 模块
def rm_tags(text):	创建 rm_tags 函数，输入参数 text 文字
re_tag = re.compile(r'<[^>]+>')	创建 re_tag 为正则表达式变量，且赋值为 '<[^>]+>'
return re_tag.sub('', text)	使用 re_tag 将 text 文字中符合正则表达式条件的字符替换成空字符串

3. 创建 read_files 函数读取 IMDb 文件目录

创建 read_files 函数读取 IMDb 文件，解压缩后的目录如图 13-11 所示。

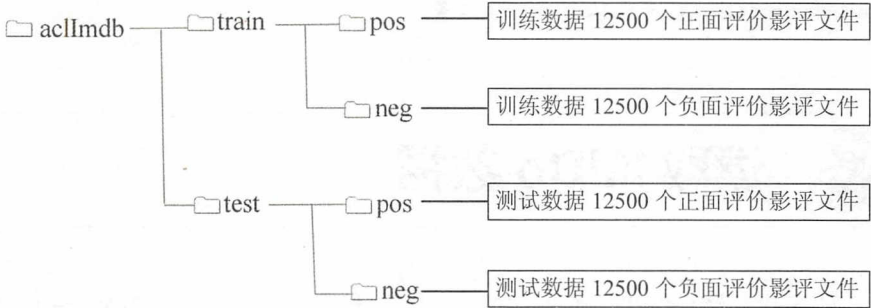


图 13-11

```
In [6]: import os
def read_files(filetype):
    path = "data/aclImdb/"
    file_list=[]

    positive_path=path + filetype+"/pos/"
    for f in os.listdir(positive_path):
        file_list+= [positive_path+f]

    negative_path=path + filetype+"/neg/"
    for f in os.listdir(negative_path):
        file_list+= [negative_path+f]

    print('read',filetype, 'files:',len(file_list))

    all_labels = ([1] * 12500 + [0] * 12500)

    all_texts = []
    for fi in file_list:
        with open(fi, encoding = 'utf8') as file_input:
            all_texts += [rm_tags(" ".join(file_input.readlines()))]

    return all_labels,all_texts
```

以上程序代码的说明见表 13-5。



表 13-5 程序代码说明

程序代码	说明
<code>import os</code>	导入 os 模块
<code>def read_files(filetype):</code>	创建 read_files 函数, 输入参数 filetype <ul style="list-style-type: none"> • 读取训练数据时 filetype 参数会传入 "train" • 读取测试数据时 filetype 参数会传入 "test"
<code>path = "data/aclImdb/"</code>	设置存取文件的路径
<code>file_list=[]</code>	创建文件列表
<code>positive_path=path + filetype+"/pos/"</code>	设置正面评价的文件目录为 positive_path
<code>for f in os.listdir(positive_path):</code> <code>file_list+= [positive_path+f]</code>	使用 for 循环将 positive_path 目录下所有的文件加入 file_list
<code>negative_path=path + filetype+"/neg/"</code>	设置负面评价的文件目录为 negative_path
<code>for f in os.listdir(negative_path):</code> <code>file_list+= [negative_path+f]</code>	使用 for 循环将 negative_path 目录下所有的文件加入 file_list
<code>print('read',filetype,</code> <code>'files:',len(file_list))</code>	显示当前读取的 filetype("train"或"test")目录下的文件个数
<code>all_labels = ([1] * 12500 +</code> <code>[0] * 12500)</code>	以下语句产生 all_labels, 因为: 前 12 500 项是正面, 所以产生 12500 项 1 的列表; 后 12 500 项是负面, 所以产生 12500 项 0 的列表
<code>all_texts = []</code>	设置 all_texts 为空列表
<code>for fi in file_list:</code> <code>with open(fi, encoding = 'utf8') as</code> <code>file_input:</code> <code>all_texts += [rm_tags (" " .</code> <code>join(file_input.readlines()))]</code>	fi 读取 file_list 所有文件: 使用 open(fi, encoding='utf8') 打开文件为 file_input; 使用 file_input.readlines() 读取文件, 并使用 join 连接所有 文件内容, 然后使用 rm_tags 删除 tag, 最后加入 all_texts list
<code>return all_labels,all_texts</code>	返回 all_labels, all_texts

4. 读取 IMDb 数据集目录

➤ 使用 read_files 函数传入参数 "train" 读取训练数据

```
In [6]: y_train,train_text=read_files("train")
        read train files: 25000
```

➤ 使用 read_files 函数传入参数 "test" 读取测试数据

```
In [7]: y_test,test_text=read_files("test")
        read test files: 25000
```

读取完成后, 将数据整理为表 13-6。

表 13-6 训练数据与测试数据说明

训练数据	train_text (文字)	0 到 12 499 项: 正面评价影评文字 12 500 到 24 999 项: 负面评价影评文字
	y_train (标签)	0 到 12 499 项: 正面评价, 全部是 1 12 500 到 24 999 项: 负面评价, 全部是 0
测试数据	test_text (文字)	0 到 12 499 项: 正面评价影评文字 12 500 到 24 999 项: 负面评价影评文字
	y_test (标签)	0 到 12 499 项: 正面评价, 所以全部是 1 12 500 到 24 999 项: 负面评价, 全部是 0

13.4 查看 IMDb 数据

读取 IMDb 数据集后, 我们就可以查看“影评文字”了。

1. 查看第 0 项“影评文字”

下面的指令用来查看第 0 项“影评文字”。

```
In [9]: train_text[0]
Out[9]: "Sure this was a remake of a 70's film, but it had the suspense and a
ction of a current film, say Breakdown. He's running, desperate to be
with his hospitalized wife, the police are the least concern. The cha
ses were very good, the part with him being cornered at a rest stop wa
s well done, the end of the movie was a great cliffhanger. This is be
tter than Bullitt, a boring movie with what, a muscle car chase that
```

2. 查看第 0 项 label 是 1, 也就是正面评价

```
In [10]: y_train[0]
Out[10]: 1
```

3. 查看第 12 501 项影评文字

```
In [12]: train_text[12501]
Out[12]: "The first scene in 'Problem Child' has a baby peeing into a nun's fa
ce. For this movie, that's witty. A nasty, mean-spirited 'comedy', it
's inept on so many levels it beggars belief. John Ritter is the kind
father who adopts the child from Hell, and kudos to him for maintaini
ng his dignity in the surrounding onslaught of one-note, annoying per
formances and puerile humour. And what the hell's Jack Warden doing i
n this mess? Slackly directed by Dennis Dugan and obnoxious in its at
```

4. 查看第 12 501 项 label 是 0, 也就是负面评价

```
In [13]: y_train[12501]
Out[13]: 0
```




13.5 建立 token

接下来将详细介绍如何建立 token 以及 token 的特性。

1. 建立 token

```
In [15]: token = Tokenizer(num_words=2000)
         token.fit_on_texts(train_text)
```

以上程序代码说明见表 13-7。

表 13-7 程序代码说明

程序代码	说明
<code>token = Tokenizer(num_words=2000)</code>	使用 Tokenizer 建立 token，输入参数 num_words=2000，也就是说我们要建立一个有 2000 个单词的字典
<code>token.fit_on_texts(train_text)</code>	读取所有的训练数据影评，按照每一个英文单词在影评中出现的次数进行排序，排序的前 2000 名的英文单词会列入字典中

2. 查看 token 读取多少文章

通过查看 token.document_count 属性就可以知道 token 读取了多少文章。

```
In [18]: print(token.document_count)

25000
```

从以上执行结果可知，token 共读取了 25 000 项影评数据。

3. 查看 token.word_index 属性

word_index 属性也是 dict 字典数据类型，其内容是每一个单词在所有文章中出现的次数的排名，出现次数最多的“the”排在第 1 位，其次是“and”排在第 2 位。

```
In [47]: print(token.word_index)

{'the': 1, 'and': 2, 'a': 3, 'of': 4, 'to': 5, 'is': 6, 'in': 7, 'it': 8, 'i': 9, 'this': 10, 'that': 11, 'was': 12, 'as': 13, 'for': 14, 'with': 15, 'movie': 16, 'but': 17, 'film': 18, 'on': 19, 'not': 20, 'you': 21, 'are': 22, 'his': 23, 'have': 24, 'be': 25, 'he': 26, 'one': 27, 'all': 28, 'at': 29, 'by': 30, 'an': 31, 'they': 32, 'who': 33, 'so': 34, 'from': 35, 'like': 36, 'her': 37, 'or': 38, 'just': 39, 'about': 40, 'it's': 41, 'out': 42, 'has': 43, 'if': 44, 'some': 45, 'are': 46, 'the': 47, 'a': 48, 'of': 49, 'to': 50, 'is': 51, 'in': 52, 'it': 53, 'i': 54, 'this': 55, 'that': 56, 'was': 57, 'as': 58, 'for': 59, 'with': 60, 'movie': 61, 'but': 62, 'film': 63, 'on': 64, 'not': 65, 'you': 66, 'are': 67, 'his': 68, 'have': 69, 'be': 70, 'he': 71, 'one': 72, 'all': 73, 'at': 74, 'by': 75, 'an': 76, 'they': 77, 'who': 78, 'so': 79, 'from': 80, 'like': 81, 'her': 82, 'or': 83, 'just': 84, 'about': 85, 'it's': 86, 'out': 87, 'has': 88, 'if': 89, 'some': 90, 'are': 91, 'the': 92, 'a': 93, 'of': 94, 'to': 95, 'is': 96, 'in': 97, 'it': 98, 'i': 99, 'this': 100, 'that': 101, 'was': 102, 'as': 103, 'for': 104, 'with': 105, 'movie': 106, 'but': 107, 'film': 108, 'on': 109, 'not': 110, 'you': 111, 'are': 112, 'his': 113, 'have': 114, 'be': 115, 'he': 116, 'one': 117, 'all': 118, 'at': 119, 'by': 120, 'an': 121, 'they': 122, 'who': 123, 'so': 124, 'from': 125, 'like': 126, 'her': 127, 'or': 128, 'just': 129, 'about': 130, 'it's': 131, 'out': 132, 'has': 133, 'if': 134, 'some': 135, 'are': 136, 'the': 137, 'a': 138, 'of': 139, 'to': 140, 'is': 141, 'in': 142, 'it': 143, 'i': 144, 'this': 145, 'that': 146, 'was': 147, 'as': 148, 'for': 149, 'with': 150, 'movie': 151, 'but': 152, 'film': 153, 'on': 154, 'not': 155, 'you': 156, 'are': 157, 'his': 158, 'have': 159, 'be': 160, 'he': 161, 'one': 162, 'all': 163, 'at': 164, 'by': 165, 'an': 166, 'they': 167, 'who': 168, 'so': 169, 'from': 170, 'like': 171, 'her': 172, 'or': 173, 'just': 174, 'about': 175, 'it's': 176, 'out': 177, 'has': 178, 'if': 179, 'some': 180, 'are': 181, 'the': 182, 'a': 183, 'of': 184, 'to': 185, 'is': 186, 'in': 187, 'it': 188, 'i': 189, 'this': 190, 'that': 191, 'was': 192, 'as': 193, 'for': 194, 'with': 195, 'movie': 196, 'but': 197, 'film': 198, 'on': 199, 'not': 200, 'you': 201, 'are': 202, 'his': 203, 'have': 204, 'be': 205, 'he': 206, 'one': 207, 'all': 208, 'at': 209, 'by': 210, 'an': 211, 'they': 212, 'who': 213, 'so': 214, 'from': 215, 'like': 216, 'her': 217, 'or': 218, 'just': 219, 'about': 220, 'it's': 221, 'out': 222, 'has': 223, 'if': 224, 'some': 225, 'are': 226, 'the': 227, 'a': 228, 'of': 229, 'to': 230, 'is': 231, 'in': 232, 'it': 233, 'i': 234, 'this': 235, 'that': 236, 'was': 237, 'as': 238, 'for': 239, 'with': 240, 'movie': 241, 'but': 242, 'film': 243, 'on': 244, 'not': 245, 'you': 246, 'are': 247, 'his': 248, 'have': 249, 'be': 250, 'he': 251, 'one': 252, 'all': 253, 'at': 254, 'by': 255, 'an': 256, 'they': 257, 'who': 258, 'so': 259, 'from': 260, 'like': 261, 'her': 262, 'or': 263, 'just': 264, 'about': 265, 'it's': 266, 'out': 267, 'has': 268, 'if': 269, 'some': 270, 'are': 271, 'the': 272, 'a': 273, 'of': 274, 'to': 275, 'is': 276, 'in': 277, 'it': 278, 'i': 279, 'this': 280, 'that': 281, 'was': 282, 'as': 283, 'for': 284, 'with': 285, 'movie': 286, 'but': 287, 'film': 288, 'on': 289, 'not': 290, 'you': 291, 'are': 292, 'his': 293, 'have': 294, 'be': 295, 'he': 296, 'one': 297, 'all': 298, 'at': 299, 'by': 300, 'an': 301, 'they': 302, 'who': 303, 'so': 304, 'from': 305, 'like': 306, 'her': 307, 'or': 308, 'just': 309, 'about': 310, 'it's': 311, 'out': 312, 'has': 313, 'if': 314, 'some': 315, 'are': 316, 'the': 317, 'a': 318, 'of': 319, 'to': 320, 'is': 321, 'in': 322, 'it': 323, 'i': 324, 'this': 325, 'that': 326, 'was': 327, 'as': 328, 'for': 329, 'with': 330, 'movie': 331, 'but': 332, 'film': 333, 'on': 334, 'not': 335, 'you': 336, 'are': 337, 'his': 338, 'have': 339, 'be': 340, 'he': 341, 'one': 342, 'all': 343, 'at': 344, 'by': 345, 'an': 346, 'they': 347, 'who': 348, 'so': 349, 'from': 350, 'like': 351, 'her': 352, 'or': 353, 'just': 354, 'about': 355, 'it's': 356, 'out': 357, 'has': 358, 'if': 359, 'some': 360, 'are': 361, 'the': 362, 'a': 363, 'of': 364, 'to': 365, 'is': 366, 'in': 367, 'it': 368, 'i': 369, 'this': 370, 'that': 371, 'was': 372, 'as': 373, 'for': 374, 'with': 375, 'movie': 376, 'but': 377, 'film': 378, 'on': 379, 'not': 380, 'you': 381, 'are': 382, 'his': 383, 'have': 384, 'be': 385, 'he': 386, 'one': 387, 'all': 388, 'at': 389, 'by': 390, 'an': 391, 'they': 392, 'who': 393, 'so': 394, 'from': 395, 'like': 396, 'her': 397, 'or': 398, 'just': 399, 'about': 400, 'it's': 401, 'out': 402, 'has': 403, 'if': 404, 'some': 405, 'are': 406, 'the': 407, 'a': 408, 'of': 409, 'to': 410, 'is': 411, 'in': 412, 'it': 413, 'i': 414, 'this': 415, 'that': 416, 'was': 417, 'as': 418, 'for': 419, 'with': 420, 'movie': 421, 'but': 422, 'film': 423, 'on': 424, 'not': 425, 'you': 426, 'are': 427, 'his': 428, 'have': 429, 'be': 430, 'he': 431, 'one': 432, 'all': 433, 'at': 434, 'by': 435, 'an': 436, 'they': 437, 'who': 438, 'so': 439, 'from': 440, 'like': 441, 'her': 442, 'or': 443, 'just': 444, 'about': 445, 'it's': 446, 'out': 447, 'has': 448, 'if': 449, 'some': 450, 'are': 451, 'the': 452, 'a': 453, 'of': 454, 'to': 455, 'is': 456, 'in': 457, 'it': 458, 'i': 459, 'this': 460, 'that': 461, 'was': 462, 'as': 463, 'for': 464, 'with': 465, 'movie': 466, 'but': 467, 'film': 468, 'on': 469, 'not': 470, 'you': 471, 'are': 472, 'his': 473, 'have': 474, 'be': 475, 'he': 476, 'one': 477, 'all': 478, 'at': 479, 'by': 480, 'an': 481, 'they': 482, 'who': 483, 'so': 484, 'from': 485, 'like': 486, 'her': 487, 'or': 488, 'just': 489, 'about': 490, 'it's': 491, 'out': 492, 'has': 493, 'if': 494, 'some': 495, 'are': 496, 'the': 497, 'a': 498, 'of': 499, 'to': 500, 'is': 501, 'in': 502, 'it': 503, 'i': 504, 'this': 505, 'that': 506, 'was': 507, 'as': 508, 'for': 509, 'with': 510, 'movie': 511, 'but': 512, 'film': 513, 'on': 514, 'not': 515, 'you': 516, 'are': 517, 'his': 518, 'have': 519, 'be': 520, 'he': 521, 'one': 522, 'all': 523, 'at': 524, 'by': 525, 'an': 526, 'they': 527, 'who': 528, 'so': 529, 'from': 530, 'like': 531, 'her': 532, 'or': 533, 'just': 534, 'about': 535, 'it's': 536, 'out': 537, 'has': 538, 'if': 539, 'some': 540, 'are': 541, 'the': 542, 'a': 543, 'of': 544, 'to': 545, 'is': 546, 'in': 547, 'it': 548, 'i': 549, 'this': 550, 'that': 551, 'was': 552, 'as': 553, 'for': 554, 'with': 555, 'movie': 556, 'but': 557, 'film': 558, 'on': 559, 'not': 560, 'you': 561, 'are': 562, 'his': 563, 'have': 564, 'be': 565, 'he': 566, 'one': 567, 'all': 568, 'at': 569, 'by': 570, 'an': 571, 'they': 572, 'who': 573, 'so': 574, 'from': 575, 'like': 576, 'her': 577, 'or': 578, 'just': 579, 'about': 580, 'it's': 581, 'out': 582, 'has': 583, 'if': 584, 'some': 585, 'are': 586, 'the': 587, 'a': 588, 'of': 589, 'to': 590, 'is': 591, 'in': 592, 'it': 593, 'i': 594, 'this': 595, 'that': 596, 'was': 597, 'as': 598, 'for': 599, 'with': 600, 'movie': 601, 'but': 602, 'film': 603, 'on': 604, 'not': 605, 'you': 606, 'are': 607, 'his': 608, 'have': 609, 'be': 610, 'he': 611, 'one': 612, 'all': 613, 'at': 614, 'by': 615, 'an': 616, 'they': 617, 'who': 618, 'so': 619, 'from': 620, 'like': 621, 'her': 622, 'or': 623, 'just': 624, 'about': 625, 'it's': 626, 'out': 627, 'has': 628, 'if': 629, 'some': 630, 'are': 631, 'the': 632, 'a': 633, 'of': 634, 'to': 635, 'is': 636, 'in': 637, 'it': 638, 'i': 639, 'this': 640, 'that': 641, 'was': 642, 'as': 643, 'for': 644, 'with': 645, 'movie': 646, 'but': 647, 'film': 648, 'on': 649, 'not': 650, 'you': 651, 'are': 652, 'his': 653, 'have': 654, 'be': 655, 'he': 656, 'one': 657, 'all': 658, 'at': 659, 'by': 660, 'an': 661, 'they': 662, 'who': 663, 'so': 664, 'from': 665, 'like': 666, 'her': 667, 'or': 668, 'just': 669, 'about': 670, 'it's': 671, 'out': 672, 'has': 673, 'if': 674, 'some': 675, 'are': 676, 'the': 677, 'a': 678, 'of': 679, 'to': 680, 'is': 681, 'in': 682, 'it': 683, 'i': 684, 'this': 685, 'that': 686, 'was': 687, 'as': 688, 'for': 689, 'with': 690, 'movie': 691, 'but': 692, 'film': 693, 'on': 694, 'not': 695, 'you': 696, 'are': 697, 'his': 698, 'have': 699, 'be': 700, 'he': 701, 'one': 702, 'all': 703, 'at': 704, 'by': 705, 'an': 706, 'they': 707, 'who': 708, 'so': 709, 'from': 710, 'like': 711, 'her': 712, 'or': 713, 'just': 714, 'about': 715, 'it's': 716, 'out': 717, 'has': 718, 'if': 719, 'some': 720, 'are': 721, 'the': 722, 'a': 723, 'of': 724, 'to': 725, 'is': 726, 'in': 727, 'it': 728, 'i': 729, 'this': 730, 'that': 731, 'was': 732, 'as': 733, 'for': 734, 'with': 735, 'movie': 736, 'but': 737, 'film': 738, 'on': 739, 'not': 740, 'you': 741, 'are': 742, 'his': 743, 'have': 744, 'be': 745, 'he': 746, 'one': 747, 'all': 748, 'at': 749, 'by': 750, 'an': 751, 'they': 752, 'who': 753, 'so': 754, 'from': 755, 'like': 756, 'her': 757, 'or': 758, 'just': 759, 'about': 760, 'it's': 761, 'out': 762, 'has': 763, 'if': 764, 'some': 765, 'are': 766, 'the': 767, 'a': 768, 'of': 769, 'to': 770, 'is': 771, 'in': 772, 'it': 773, 'i': 774, 'this': 775, 'that': 776, 'was': 777, 'as': 778, 'for': 779, 'with': 780, 'movie': 781, 'but': 782, 'film': 783, 'on': 784, 'not': 785, 'you': 786, 'are': 787, 'his': 788, 'have': 789, 'be': 790, 'he': 791, 'one': 792, 'all': 793, 'at': 794, 'by': 795, 'an': 796, 'they': 797, 'who': 798, 'so': 799, 'from': 800, 'like': 801, 'her': 802, 'or': 803, 'just': 804, 'about': 805, 'it's': 806, 'out': 807, 'has': 808, 'if': 809, 'some': 810, 'are': 811, 'the': 812, 'a': 813, 'of': 814, 'to': 815, 'is': 816, 'in': 817, 'it': 818, 'i': 819, 'this': 820, 'that': 821, 'was': 822, 'as': 823, 'for': 824, 'with': 825, 'movie': 826, 'but': 827, 'film': 828, 'on': 829, 'not': 830, 'you': 831, 'are': 832, 'his': 833, 'have': 834, 'be': 835, 'he': 836, 'one': 837, 'all': 838, 'at': 839, 'by': 840, 'an': 841, 'they': 842, 'who': 843, 'so': 844, 'from': 845, 'like': 846, 'her': 847, 'or': 848, 'just': 849, 'about': 850, 'it's': 851, 'out': 852, 'has': 853, 'if': 854, 'some': 855, 'are': 856, 'the': 857, 'a': 858, 'of': 859, 'to': 860, 'is': 861, 'in': 862, 'it': 863, 'i': 864, 'this': 865, 'that': 866, 'was': 867, 'as': 868, 'for': 869, 'with': 870, 'movie': 871, 'but': 872, 'film': 873, 'on': 874, 'not': 875, 'you': 876, 'are': 877, 'his': 878, 'have': 879, 'be': 880, 'he': 881, 'one': 882, 'all': 883, 'at': 884, 'by': 885, 'an': 886, 'they': 887, 'who': 888, 'so': 889, 'from': 890, 'like': 891, 'her': 892, 'or': 893, 'just': 894, 'about': 895, 'it's': 896, 'out': 897, 'has': 898, 'if': 899, 'some': 900, 'are': 901, 'the': 902, 'a': 903, 'of': 904, 'to': 905, 'is': 906, 'in': 907, 'it': 908, 'i': 909, 'this': 910, 'that': 911, 'was': 912, 'as': 913, 'for': 914, 'with': 915, 'movie': 916, 'but': 917, 'film': 918, 'on': 919, 'not': 920, 'you': 921, 'are': 922, 'his': 923, 'have': 924, 'be': 925, 'he': 926, 'one': 927, 'all': 928, 'at': 929, 'by': 930, 'an': 931, 'they': 932, 'who': 933, 'so': 934, 'from': 935, 'like': 936, 'her': 937, 'or': 938, 'just': 939, 'about': 940, 'it's': 941, 'out': 942, 'has': 943, 'if': 944, 'some': 945, 'are': 946, 'the': 947, 'a': 948, 'of': 949, 'to': 950, 'is': 951, 'in': 952, 'it': 953, 'i': 954, 'this': 955, 'that': 956, 'was': 957, 'as': 958, 'for': 959, 'with': 960, 'movie': 961, 'but': 962, 'film': 963, 'on': 964, 'not': 965, 'you': 966, 'are': 967, 'his': 968, 'have': 969, 'be': 970, 'he': 971, 'one': 972, 'all': 973, 'at': 974, 'by': 975, 'an': 976, 'they': 977, 'who': 978, 'so': 979, 'from': 980, 'like': 981, 'her': 982, 'or': 983, 'just': 984, 'about': 985, 'it's': 986, 'out': 987, 'has': 988, 'if': 989, 'some': 990, 'are': 991, 'the': 992, 'a': 993, 'of': 994, 'to': 995, 'is': 996, 'in': 997, 'it': 998, 'i': 999, 'this': 1000, 'that': 1001, 'was': 1002, 'as': 1003, 'for': 1004, 'with': 1005, 'movie': 1006, 'but': 1007, 'film': 1008, 'on': 1009, 'not': 1010, 'you': 1011, 'are': 1012, 'his': 1013, 'have': 1014, 'be': 1015, 'he': 1016, 'one': 1017, 'all': 1018, 'at': 1019, 'by': 1020, 'an': 1021, 'they': 1022, 'who': 1023, 'so': 1024, 'from': 1025, 'like': 1026, 'her': 1027, 'or': 1028, 'just': 1029, 'about': 1030, 'it's': 1031, 'out': 1032, 'has': 1033, 'if': 1034, 'some': 1035, 'are': 1036, 'the': 1037, 'a': 1038, 'of': 1039, 'to': 1040, 'is': 1041, 'in': 1042, 'it': 1043, 'i': 1044, 'this': 1045, 'that': 1046, 'was': 1047, 'as': 1048, 'for': 1049, 'with': 1050, 'movie': 1051, 'but': 1052, 'film': 1053, 'on': 1054, 'not': 1055, 'you': 1056, 'are': 1057, 'his': 1058, 'have': 1059, 'be': 1060, 'he': 1061, 'one': 1062, 'all': 1063, 'at': 1064, 'by': 1065, 'an': 1066, 'they': 1067, 'who': 1068, 'so': 1069, 'from': 1070, 'like': 1071, 'her': 1072, 'or': 1073, 'just': 1074, 'about': 1075, 'it's': 1076, 'out': 1077, 'has': 1078, 'if': 1079, 'some': 1080, 'are': 1081, 'the': 1082, 'a': 1083, 'of': 1084, 'to': 1085, 'is': 1086, 'in': 1087, 'it': 1088, 'i': 1089, 'this': 1090, 'that': 1091, 'was': 1092, 'as': 1093, 'for': 1094, 'with': 1095, 'movie': 1096, 'but': 1097, 'film': 1098, 'on': 1099, 'not': 1100, 'you': 1101, 'are': 1102, 'his': 1103, 'have': 1104, 'be': 1105, 'he': 1106, 'one': 1107, 'all': 1108, 'at': 1109, 'by': 1110, 'an': 1111, 'they': 1112, 'who': 1113, 'so': 1114, 'from': 1115, 'like': 1116, 'her': 1117, 'or': 1118, 'just': 1119, 'about': 1120, 'it's': 1121, 'out': 1122, 'has': 1123, 'if': 1124, 'some': 1125, 'are': 1126, 'the': 1127, 'a': 1128, 'of': 1129, 'to': 1130, 'is': 1131, 'in': 1132, 'it': 1133, 'i': 1134, 'this': 1135, 'that': 1136, 'was': 1137, 'as': 1138, 'for': 1139, 'with': 1140, 'movie': 1141, 'but': 1142, 'film': 1143, 'on': 1144, 'not': 1145, 'you': 1146, 'are': 1147, 'his': 1148, 'have': 1149, 'be': 1150, 'he': 1151, 'one': 1152, 'all': 1153, 'at': 1154, 'by': 1155, 'an': 1156, 'they': 1157, 'who': 1158, 'so': 1159, 'from': 1160, 'like': 1161, 'her': 1162, 'or': 1163, 'just': 1164, 'about': 1165, 'it's': 1166, 'out': 1167, 'has': 1168, 'if': 1169, 'some': 1170, 'are': 1171, 'the': 1172, 'a': 1173, 'of': 1174, 'to': 1175, 'is': 1176, 'in': 1177, 'it': 1178, 'i': 1179, 'this': 1180, 'that': 1181, 'was': 1182, 'as': 1183, 'for': 1184, 'with': 1185, 'movie': 1186, 'but': 1187, 'film': 1188, 'on': 1189, 'not': 1190, 'you': 1191, 'are': 1192, 'his': 1193, 'have': 1194, 'be': 1195, 'he': 1196, 'one': 1197, 'all': 1198, 'at': 1199, 'by': 1200, 'an': 1201, 'they': 1202, 'who': 1203, 'so': 1204, 'from': 1205, 'like': 1206, 'her': 1207, 'or': 1208, 'just': 1209, 'about': 1210, 'it's': 1211, 'out': 1212, 'has': 1213, 'if': 1214, 'some': 1215, 'are': 1216, 'the': 1217, 'a': 1218, 'of': 1219, 'to': 1220, 'is': 1221, 'in': 1222, 'it': 1223, 'i': 1224, 'this': 1225, 'that': 1226, 'was': 1227, 'as': 1228, 'for': 1229, 'with': 1230, 'movie': 1231, 'but': 1232, 'film': 1233, 'on': 1234, 'not': 1235, 'you': 1236, 'are': 1237, 'his': 1238, 'have': 1239, 'be': 1240, 'he': 1241, 'one': 1242, 'all': 1243, 'at': 1244, 'by': 1245, 'an': 1246, 'they': 1247, 'who': 1248, 'so': 1249, 'from': 1250, 'like': 1251, 'her': 1252, 'or': 1253, 'just': 1254, 'about': 1255, 'it's': 1256, 'out': 1257, 'has': 1258, 'if': 1259, 'some': 1260, 'are': 1261, 'the': 1262, 'a': 1263, 'of': 1264, 'to': 1265, 'is': 1266, 'in': 1267, 'it': 1268, 'i': 1269, 'this': 1270, 'that': 1271, 'was': 1272, 'as': 1273, 'for': 1274, 'with': 1275, 'movie': 1276, 'but': 1277, 'film': 1278, 'on': 1279, 'not': 1280, 'you': 1281, 'are': 1282, 'his': 1283, 'have': 1284, 'be': 1285, 'he': 1286, 'one': 1287, 'all': 1288, 'at': 1289, 'by': 1290, 'an': 1291, 'they': 1292, 'who': 1293, 'so': 1294, 'from': 1295, 'like': 1296, 'her': 1297, 'or': 1298, 'just': 1299, 'about': 1300, 'it's': 1301, 'out': 1302, 'has': 1303, 'if': 1304, 'some': 1305, 'are': 1306, 'the': 1307, 'a': 1308, 'of': 1309, 'to': 1310, 'is': 1311, 'in': 1312, 'it': 1313, 'i': 1314, 'this': 1315, 'that': 1316, 'was': 1317, 'as': 1318, 'for': 1319, 'with': 1320, 'movie': 1321, 'but': 1322, 'film': 1323, 'on': 1324, 'not': 1325, 'you': 1326, 'are': 1327, 'his': 1328, 'have': 1329, 'be': 1330, 'he': 1331, 'one': 1332, 'all': 1333, 'at': 1334, 'by': 1335, 'an': 1336, 'they': 1337, 'who': 1338, 'so': 1339, 'from': 1340, 'like': 1341, 'her': 1342, 'or': 1343, 'just': 1344, 'about': 1345, 'it's': 1346, 'out': 1347, 'has': 1348, 'if': 1349, 'some': 1350, 'are': 1351, 'the': 1352, 'a': 1353, 'of': 1354, 'to': 1355, 'is': 1356, 'in': 1357, 'it': 1358, 'i': 1359, 'this': 1360, 'that': 1361, 'was': 1362, 'as': 1363, 'for': 1364, 'with': 1365, 'movie': 1366, 'but': 1367, 'film': 1368, 'on': 1369, 'not': 1370, 'you': 1371, 'are': 1372, 'his': 1373, 'have': 1374, 'be': 1375, 'he': 1376, 'one': 1377, 'all': 1378, 'at': 1379, 'by': 1380, 'an': 1381, 'they': 1382, 'who': 1383, 'so': 1384, 'from': 1385, 'like': 1386, 'her': 1387, 'or': 1388, 'just': 1389, 'about': 1390, 'it's': 1391, 'out
```

13.6 使用 token 将“影评文字”转换成“数字列表”

建立 token 字典后，我们就可以使用 token.word_index 字典将文字转换为“数字列表”。

1. 使用 token.texts_to_sequences 将“影评文字”转换为“数字列表”

下面的指令使用 token.texts_to_sequences 将训练数据与测试数据的“影评文字”转换成“数字列表”，将 train_text 转换为 x_train_seq，并将 test_text 转换为 x_test_seq。

```
In [31]: x_train_seq = token.texts_to_sequences(train_text)
        x_test_seq = token.texts_to_sequences(test_text)
```

2. 查看转换为 sequences 之后的结果

先使用下面的指令来查看第 0 项“影评文字”与“数字列表”。

```
In [32]: print(train_text[0])
```

Sure this was a remake of a 70's film, but it had the suspense and action of a current film, say Breakdown. He's running, desperate to be with his hospitalized wife, the police are the least concern. The chases were very good, the part with him being cornered at a rest stop was well done, the end of the movie was a great cliffhanger. This is better than Bullitt, a boring movie with what, a muscle car chase that was filmed badly? Vigo's character knew what he had to do to escape Johnny Law, few movies had the effects-night vision, CB radio-okay I forgot the name of the movie, guy has 76'Caddy souped up, toys with guy he upset. The ending is great, you can't tell if he fakes his suicide or not, a very good did-he-make-it-or-not.

```
In [33]: print(x_train_seq[0])
```

[248, 10, 12, 3, 1031, 4, 3, 1678, 18, 17, 8, 65, 1, 833, 2, 201, 4, 3, 1998, 18, 131, 236, 610, 1664, 5, 26, 15, 22, 318, 1, 563, 23, 1, 221, 1, 69, 51, 48, 1, 168, 15, 85, 29, 3, 358, 567, 12, 67, 220, 1, 125, 4, 1, 16, 12, 3, 76, 10, 6, 124, 70, 3, 367, 16, 15, 47, 3, 524, 1317, 11, 12, 798, 928, 105, 701, 47, 25, 65, 5, 78, 5, 1069, 1783, 1, 153, 167, 98, 65, 1, 301, 312, 1777, 1852, 878, 9, 1, 404, 4, 1, 16, 234, 43, 52, 15, 234, 25, 1, 273, 6, 76, 21, 188, 371, 44, 25, 22, 16, 97, 38, 20, 3, 51, 48, 118, 25, 93, 8, 38, 20]

Sure 转换成 248、this 转换成 10、was 转换成 12

以上“影评文字”已经转换为“数字列表”，例如前 3 个单词中的 Sure 转换成 248、this 转换成 10、was 转换成 12。

13.7 让转换后的数字长度相同

因为每一则“影评文字”的单词数都不固定，例如有些可能有 170 个单词，有些有 80 个



单词。转换成“数字列表”的数字个数也就不固定。因为后续要将“数字列表”转为“向量列表”，并送入深度学习模型进行训练，所以长度必须固定。如何让所有“数字列表”长度都固定呢？其实方法很简单，就是截长补短。

例如，我们要将“数字列表”的长度都设置为 100。

- 如果数字列表的长度为 126，就将前面的 26 个数字截去，这样就变成长度为 100 的“数字列表”。
- 如果数字列表的长度是 59，就在前面补上 41 个“0”，这样就变成长度为 100 的“数字列表”。

1. 使用 sequence.pad_sequences()方法截长补短

在前面步骤中产生的“数字列表”：x_train_seq、x_test_seq，使用 sequence.pad_sequences 进行截长补短，让每一个“数字列表”长度都是 100。

```
In [39]: x_train = sequence.pad_sequences(x_train_seq, maxlen=100)
         x_test = sequence.pad_sequences(x_test_seq, maxlen=100)
```

2. “影评文字”转成“数字列表”后，长度大于 100 的处理方式

下面的第 0 项“影评文字”转成“数字列表”后，长度为 126，大于 100，使用 pad_sequences 处理之后会截去“数字列表”前面的数字，使处理后的长度为 100。

➤ 显示第 0 项“数字列表”

```
In [25]: print('before pad_sequences length=', len(x_train_seq[0]))
         print(x_train_seq[0])
```

```
before pad_sequences length= 126
[248, 10, 12, 3, 1029, 4, 3, 1691, 18, 17, 8, 65, 1, 835, 2, 202, 4,
 3, 18, 131, 236, 616, 1676, 5, 25, 15, 23, 318, 1, 564, 22, 1, 218, 1,
 67, 51, 48, 1, 169, 15, 86, 29, 3, 356, 566, 12, 69, 220, 1, 126, 4,
 1, 16, 12, 3, 83, 10, 6, 124, 70, 3, 354, 16, 15, 47, 3, 515, 1309,
 11, 12, 810, 907, 105, 693, 47, 26, 65, 5, 78, 5, 1084, 1800, 1159, 1
 67, 98, 65, 1, 297, 310, 1767, 1871, 869, 9, 1, 399, 4, 1, 16, 229, 4
 3, 52, 15, 229, 26, 1, 273, 6, 83, 21, 187, 371, 44, 26, 23, 1711, 38
 20, 3, 51, 48, 118, 26, 93, 8, 38, 20]
```

长度大于100，必须要截去前面的数字

截去前面的数字

➤ 显示第 0 项“数字列表”，经过 pad_sequences 处理后的内容

```
In [26]: print('after pad_sequences length=', len(x_train[0]))
         print(x_train[0])
```

```
after pad_sequences length= 100
[ 23 318 1 564 22 1 218 1 67 51 48 1 169 1
 5 86
 29 3 356 566 12 69 220 1 126 4 1 16 12
 3 83
 10 6 124 70 3 354 16 15 47 3 515 1309 11 1
 2 810
 907 105 693 47 26 65 5 78 5 1084 1800 1159 167 9
 8 65
 1 297 310 1767 1871 869 9 1 399 4 1 16 229 4
 3 52
 15 229 26 1 273 6 83 21 187 371 44 26 23 171
 1 38
 20 3 51 48 118 26 93 8 38 20]
```

截去前面的数字后剩下的数字列表

从以上执行结果可知，23 之前的数字都被截去了，“数字列表”的长度变为 100。

3. “影评文字”转成“数字列表”后，长度小于100的处理方式

下面第一项“影评文字”转成“数字列表”后，长度是 59，小于 100，使用 `pad_sequences` 处理之后会在“数字列表”前面，填上 41 个数字 0，使处理后的长度为 100。

```
In [28]: print('before pad_sequences length=', len(x_train_seq[1]))
         print(x_train_seq[1])
```

before pad_sequences length=59 长度小于100, 前面的数字会填上数字0

[90, 20, 13, 48, 13, 1, 82, 16, 17, 90, 3, 48, 1151, 16, 90, 43, 48, 515, 1309, 135, 19, 1, 1029, 4, 10, 16, 46, 3, 61, 14, 22, 627, 5, 12 74, 698, 13, 23, 265, 5, 5, 1, 495, 4, 23, 318, 1, 273, 6, 83, 39, 3, 48, 1247, 16, 5, 102, 7, 57, 648]	
--	--

```
In [29]: print('after pad_sequences length=', len(x_train[1]))
          print(x_train[1])
```

前面会填上数字0

after pad sequences length= 100

[0	0	0	0	0	0	0	0	0	0	0	0	0
0	0												
	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0												
	0	0	0	0	0	0	0	0	0	0	0	90	20
3	48											1	
	13	1	82	16	17	90	3	48	1151	16	90	43	48
5	1309											51	
	135	19	1	1029	4	10	16	46	3	61	14	22	627
5	1274												
	698	13	23	265	5	5	1	495	4	23	318	1	273
6	83												
	39	3	48	1247	16	5	102	7	57	648]			

13.8 结论

在本章中，我们介绍了如何下载与读取 IMDB 数据集，并且完成了数据预处理。在下一章中，我们可以使用 Keras 建立多层感知器、RNN、LSTM 的模型，并训练模型，然后使用训练完成的模型进行预测。

第14章

Keras建立MLP、RNN、LSTM模型进行IMDb情感分析

在上一章我们已经完成了 IMDb 数据集的预处理。在本章我们使用 Keras 建立多层感知器、递归神经网络、长短时记忆模型，进行 IMDb 情感分析，并训练模型、进行预测，最后产生预测结果（“正面评价”或“负面评价”）。

14.1 建立多层感知器模型进行 IMDB 情感分析

首先，我们将建立多层感知器模型，整理如图 14-1 所示。

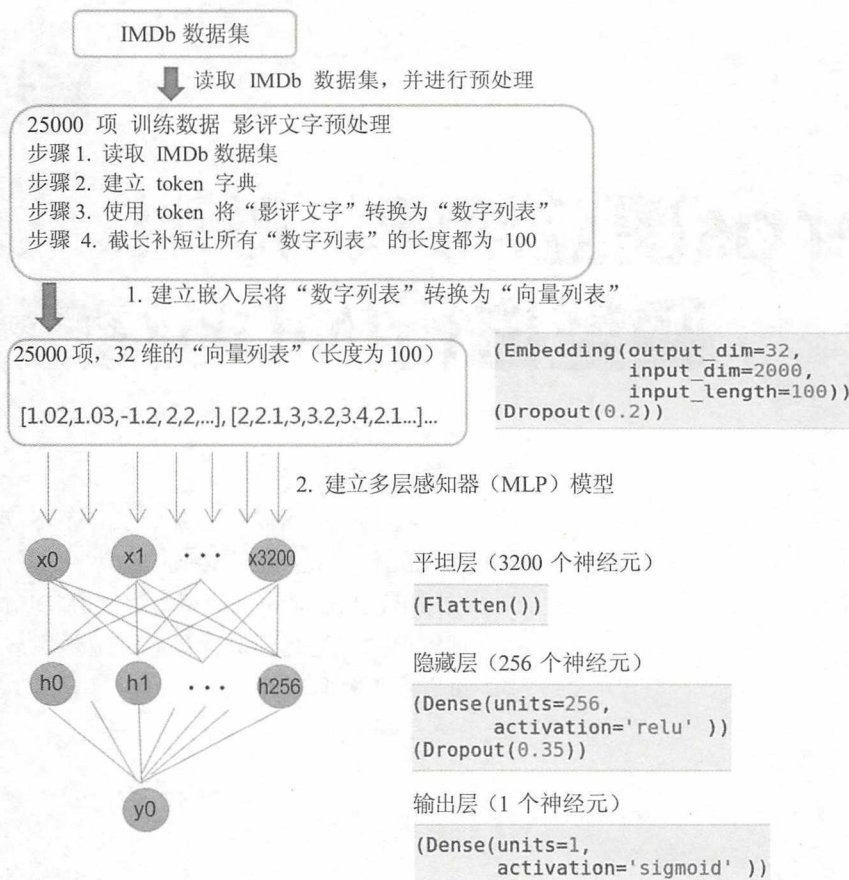


图 14-1

图 14-1 的步骤 1~4 文字的预处理已经在第 13 章介绍了。本章将建立：

- (1) **嵌入层** 将“数字列表”转换为“向量列表”。
- (2) **多层感知器** 使用多层感知器模型处理“向量列表”。

- **平坦层**：共有 3200 个神经元，因为原本“数字列表”每一项有 100 个数字，每一个数字转换为 32 维的向量，所以转换为平坦层的神经元有 3200 个 ($32 \times 100 = 3200$)。
- **隐藏层**：共有 256 个神经元。
- **输出层**：只有 1 个神经元，输出 1 代表正面评价，0 代表负面评价。

本章完整的程序代码可参考范例程序 Keras_Imdb_MLP.ipynb。范例程序下载与安装可参

考本书附录 A。

14.2 数据预处理

关于数据预处理，在第 13 章已经详细介绍过了，本章整理其中的主要步骤如下。

步骤01 导入所需模块。

```
In [1]: from keras.datasets import imdb
        from keras.preprocessing import sequence
        from keras.preprocessing.text import Tokenizer

        Using TensorFlow backend.
```

步骤02 读取 IMDb 数据集目录。

使用第 13 章所介绍的 `read_files` 函数读取 IMDb 数据，细节请参考第 13 章。

➤ 读取训练数据

```
In [6]: y_train, train_text = read_files("train")

        read train files: 25000
```

➤ 读取测试数据

```
In [7]: y_test, test_text = read_files("test")

        read test files: 25000
```

步骤03 建立 token。

```
In [9]: token = Tokenizer(num_words=2000)
        token.fit_on_texts(train_text)
```

步骤04 将“影评文字”转换成“数字列表”。

```
In [11]: x_train_seq = token.texts_to_sequences(train_text)
        x_test_seq = token.texts_to_sequences(test_text)
```

步骤05 截长补短让所有“数字列表”的长度都为 100。

```
In [13]: x_train = sequence.pad_sequences(x_train_seq, maxlen=100)
        x_test = sequence.pad_sequences(x_test_seq, maxlen=100)
```

以上“数字列表”都是由“影评文字”转换而来的，并且已经截长补短每一项“数字列表”，因而列表中都正好是 100 个数字。

步骤06 数据预处理完成后的数据整理。

以上数据预处理完成后产生训练数据与测试数据。

- **训练数据：**我们将送入各种深度学习模型进行训练。

训练数据	x_train (feature)	0 到 12499 项：正面评价的“数字列表”； 12500 到 24999 项：负面评价的“数字列表”
	y_train (label)	0 到 12499 项：正面评价，全部是 1； 12500 到 24999 项：负面评价，全部是 0

- **测试数据：**可以用于评估深度学习模型的准确率，并进行预测。

测试数据	x_test (feature)	0 到 12499 项：正面评价的“数字列表”； 12500 到 24999 项：负面评价的“数字列表”
	y_test (label)	0 到 12499 项：正面评价，全部是 1； 12500 到 24999 项：负面评价，全部是 0

14.3 加入嵌入层

Keras 提供了嵌入层可以将“数字列表”转换为“向量列表”。关于词嵌入自然语言处理技术的说明可参考第 13 章。

步骤01 导入所需模块。

```
In [13]: from keras.models import Sequential
         from keras.layers.core import Dense, Dropout, Activation, Flatten
         from keras.layers.embeddings import Embedding
```

步骤02 建立模型。

使用下面的程序代码建立一个线性堆叠模型，后续只需要将各个神经网络层加入模型即可。

```
In [14]: model = Sequential()
```

步骤03 将“嵌入层”加入模型。

使用下面的程序代码将“嵌入层”加入模型。

```
In [15]: model.add(Embedding(output_dim=32,
                             input_dim=2000,
                             input_length=100))
         model.add(Dropout(0.2))
```

➤ 建立嵌入层需输入表 14-1 中的参数。

表 14-1 建立嵌入层需输入的参数

参数	参数说明
output_dim=32	输出的维数为 32，因为我们希望将“数字列表”转换为 32 维的向量
input_dim=2000	输入的维数是 2000，因为之前建立的字典有 2000 个单词
input_length=100	因为“数字列表”每一项有 100 个数字

➤ 加入 Dropout 层以避免过度拟合

Dropout(0.2)的功能是，每次训练迭代时会随机地在神经网络中放弃 20%的神经元，以避免过度拟合。

14.4 建立多层感知器模型

嵌入层转换为“向量列表”后，就可以使用各种深度学习模型进行训练与预测了。本节先介绍建立多层感知器模型。

步骤01 将“平坦层”加入模型。

使用下面的程序代码将“平坦层”加入模型。因为“数字列表”每一项有 100 个数字，每一个数字转换为 32 维的向量，所以转换为平坦层的神经元有 3200 个 ($100 \times 32 = 3200$)。

```
In [16]: model.add(Flatten())
```

步骤02 将“隐藏层”加入模型。

```
In [18]: model.add(Dense(units=256,
                           activation='relu' ))
        model.add(Dropout(0.35))
```

➤ 建立“隐藏层”使用 Dense 神经网络层，需输入表 14-2 中的参数。

表 14-2 建立“隐藏层”使用 Dense 神经网络层需输入的参数

参数	参数说明
unit=256	隐藏层共有 256 个神经元
Activation='relu'	定义激活函数 ReLU

➤ 加入 Dropout 层以避免过度拟合

Dropout(0.35)的功能是，每次训练迭代时会随机地在神经网络中放弃 25%的神经元，以避免过度拟合。

步骤03 将“输出层”加入模型。

```
In [19]: model.add(Dense(units=1,
                        activation='sigmoid' ))
```

➤ 建立“输出层”使用 Dense 神经网络层，需输入表 14-3 中的参数。

表 14-3 建立“输出层”使用 Dense 神经网络层需输入的参数

参数	参数说明
Unit=1	输出层只有 1 个神经元，输出 1 代表正面评价，0 代表负面评价
Activation='sigmoid'	定义激活函数 Sigmoid

步骤04 查看模型的摘要。

可以使用下列指令来查看模型的摘要。

```
In [19]: model.summary()
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 100, 32)	64000
dropout_1 (Dropout)	(None, 100, 32)	0
flatten_1 (Flatten)	(None, 3200)	0
dense_1 (Dense)	(None, 256)	819456
dropout_2 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 1)	257
Total params: 883,713.0		
Trainable params: 883,713.0		
Non-trainable params: 0.0		

嵌入层

平坦层

隐藏层

输出层

14.5 训练模型

当我们建立深度学习模型后，就可以使用反向传播算法进行训练。可参考第 2 章有关使用反向传播算法进行训练的说明。

1. 定义训练方式

在训练模型之前，我们必须使用 `compile` 方法对训练模型进行设置，如下列指令：

```
In [20]: model.compile(loss='binary_crossentropy',
                    optimizer='adam',
                    metrics=['accuracy'])
```

`compile` 方法需输入 3 个参数：loss、optimizer 和 metrics（对这 3 个参数的解释说明可参



考下 7.4 节)。

2. 开始训练

```
In [21]: train_history = model.fit(x_train, y_train, batch_size=100,
                                   epochs=10, verbose=2,
                                   validation_split=0.2)
```

使用 `model.fit` 进行训练，训练过程会存储在 `train_history` 变量中，这个训练需输入下列参数。

(1) 输入训练数据参数

- `x=x_train`: features (“数字列表”)。
- `y=y_train`: 测试数据的标签 label (影评的真实值，正向: 1, 负向: 0)。

(2) 设置训练与验证数据的比例

训练之前 Keras 会自动将数据分成: 80% 作为训练数据, 20% 作为验证数据。因为全部数据有 25 000 项, 所以分成: $25\,000 \times 0.8 = 20\,000$ 作为训练数据, $25\,000 \times 0.2 = 5\,000$ 作为验证数据。

(3) 设置训练周期数与批次的项数

- `epochs=10`: 执行 10 个训练周期。
- `batch_size=100`: 每一批次 100 项数据。

(4) 设置显示训练过程

- `verbose=2`: 显示训练过程。

上面的程序代码共执行了 10 个训练周期，每一个训练周期中执行下列功能：

- 使用 20 000 项训练数据进行训练，分为每一批次 100 项，所以大约分为 200 个批次 ($20\,000/100=200$) 进行训练。
- Epoch (训练周期) 训练完成后，会计算这个训练周期的准确率与误差，并且在 `train_history` 中新增一项数据记录。



➤ 以上程序代码执行后的结果如图 14-2 所示。

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
12s - loss: 0.4681 - acc: 0.7647 - val_loss: 0.5636 - val_acc: 0.7330
Epoch 2/10
13s - loss: 0.2607 - acc: 0.8923 - val_loss: 0.5205 - val_acc: 0.7664
Epoch 3/10
12s - loss: 0.1587 - acc: 0.9419 - val_loss: 0.6557 - val_acc: 0.7582
Epoch 4/10
12s - loss: 0.0872 - acc: 0.9694 - val_loss: 0.9137 - val_acc: 0.7240
Epoch 5/10
12s - loss: 0.0482 - acc: 0.9835 - val_loss: 1.0317 - val_acc: 0.7532
Epoch 6/10
12s - loss: 0.0364 - acc: 0.9864 - val_loss: 1.0940 - val_acc: 0.7528
Epoch 7/10
13s - loss: 0.0288 - acc: 0.9899 - val_loss: 1.2512 - val_acc: 0.7442
Epoch 8/10
17s - loss: 0.0282 - acc: 0.9900 - val_loss: 1.2747 - val_acc: 0.7542
Epoch 9/10
18s - loss: 0.0249 - acc: 0.9909 - val_loss: 1.4221 - val_acc: 0.7358
Epoch 10/10
12s - loss: 0.0222 - acc: 0.9920 - val_loss: 1.6406 - val_acc: 0.7174
```

图 14-2

从以上执行界面可知，共执行了 10 个训练周期，还可以发现误差越来越小，准确率越来越高。

14.6 评估模型准确率

之前我们已经完成了模型的训练，现在要使用 test 测试数据集，评估模型的准确率。下面的程序代码用来评估模型的准确率。

```
In [25]: scores = model.evaluate(x_test, y_test, verbose=1)
        scores[1]
24928/25000 [=====.] - ETA: 0s - ETA: 14s
Out[25]: 0.8081599999999999
```

从以上执行结果可知准确率是 0.80。以上程序代码说明见表 14-4。

表 14-4 程序代码说明

程序代码	说明
scores = model.evaluate(使用 model.evaluate 评估模型的准确率，评估后的准确率会存储在 scores 中
x_test,	测试数据的特征值 features（“数字列表”，每一项有 100 个数字）
y_test)	测试数据的标签 label（影评的真实值，正向：1，负向：0）
scores[1]	显示准确率

14.7 进行预测

在前面的步骤中，我们建立了模型，并且完成了模型的训练，准确率达到还可以接受的 0.80，接下来我们将使用这个模型进行预测。

步骤01 执行预测。

我们可以用下列指令执行预测：

```
In [30]: predict=model.predict_classes(x_test)
24992/25000 [=====>.] - ETA: 0s
```

以上程序代码使用 `model.predict_classes` 进行预测，输入参数：测试数据的特征值 `features`（数字列表）。

步骤02 预测结果。

我们可以用下列指令来查看预测结果的前 10 项数据。

```
In [31]: predict[:10]
Out[31]: array([[1],
                [1],
                [1],
                [0],
                [1],
                [1],
                [1],
                [1],
                [1],
                [1]], dtype=int32)
```

从以上执行结果可知，0 代表负面评价，1 代表正面评价。

步骤03 使用一维数组查看预测结果。

上一步的执行结果 `predict` 是二维数组，我们可以使用 `reshape` 把它转换为一维数组 `predict_class`。

```
In [28]: predict_classes=predict.reshape(-1)
predict_classes[:10]
Out[28]: array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=int32)
```

14.8 查看测试数据预测结果

之前的预测结果是 0 与 1，我们将创建 `display_test_Sentiment` 函数，可以显示负面评价或正面评价。

步骤01 创建 display_test_Sentiment 函数。

创建 display_test_Sentiment 函数如下：

```
In [57]: SentimentDict={1:'正面的',0:'负面的'}
def display_test_Sentiment(i):
    print(test_text[i])
    print('label真实值:',SentimentDict[y_test[i]],
          '预测结果:',SentimentDict[predict_classes[i]])
```

程序代码说明见表 14-5。

表 14-5 程序代码说明

程序代码	说明
SentimentDict={1:' 正面的 ',0:' 负面的 '}	定义字典 1: 正面的, 0: 负面的
def display_test_Sentiment(i):	创建 display_test_Sentiment 函数, 输入参数 i 指定要显示第几项数据
print(test_text[i])	显示“影评文字”
print(' 标 签 label:', SentimentDict[y_test[i]],	显示影评的真实值 y_test[i], 并使用 SentimentDict 字典转换 显示 1: '正面的', 0: '负面的'
' 预 测 结 果 :', SentimentDict[predict_classes[i]])	显示预测结果 predict_classes[i], 并使用 SentimentDict 字典转 换显示 1: '正面的', 0: '负面的'

步骤02 显示第 2 项预测结果。

以下使用 display_test_Sentiment 函数来显示预测结果。

```
In [58]: display_test_Sentiment(2)

When you think of golf movies, you think of Caddyshack, but what i
f there are kids around? Go right to this movie! Disney uses is pr
oved formula to make a movie that the adults and the kids will enj
oy. The acting in this movie, in my opinion, is quite good and the
leading cast, for the most part, is very young! This movie is also
suprsingly filmed very well and unique, seeing all the angles of t
he golf game. I think this movie should be up for some academy awa
rds for film editing or something like that because the entire flo
w of the film is top notch. Though the ending might be a little pr
edictable, the movie does well on its own! It also shows that you
do not need swearing, nudity, or violence to make a great golf mov
ie!
label真实值: 正面的    预测结果: 正面的
```

从以上执行结果可知, 第 2 项数据真实值是正面的, 预测结果也是正面的。

步骤03 显示第 12 502 项预测结果。

```
In [67]: display_test_Sentiment 12502
```

I'm almost embarrassed to admit to seeing CALIGULA twice. The problems with the production are almost too numerous to mention. The script is sub-standard (it's easy to see why Vidal tried to disown it). The direction is worse. Most of the movie consists of long shots inter cut with close-ups interspersed with cross cuts of mostly un-erotic porn (more prevalent obviously in the "uncut" version). The cinematography is especially sub par, giving the whole production a cheap washed-out look that undermines some of the elaborate set designs. The movie should've looked a whole lot better. The overall concept of placing name actors in what would've easily been an X-rated movie (Guccione called it "paganography") wears thin after the first hour after Peter O'Toole and John Guilgud exit. Bob Guccione obviously lavished a lot of bucks on this but it all seems like a big waste. If you want a far better understanding of the Roman Empire in the 1st Century watch the mid-79's BBC production of I, CLAUDIUS instead... and if you want porn, jeeze-Louise, look somewhere else.

label真实值: 负面的 预测结果: 负面的

从以上执行结果可知, 第 12502 项数据真实值是负面的, 预测结果也是负面的。

14.9 查看《美女与野兽》的影评

之前的预测使用的是 IMDb 数据集的影评文字, 接下来将使用热门电影《美女与野兽》的影评文字进行预测。

步骤01 查看美女与野兽的影评。

可以在下列网址查看《美女与野兽》的影评:

<http://www.imdb.com/title/tt2771200/reviews>

在影评页面可以筛选影评, 我们选择 Chronological, 即按时间顺序进行排序, 如图 14-3 所示。

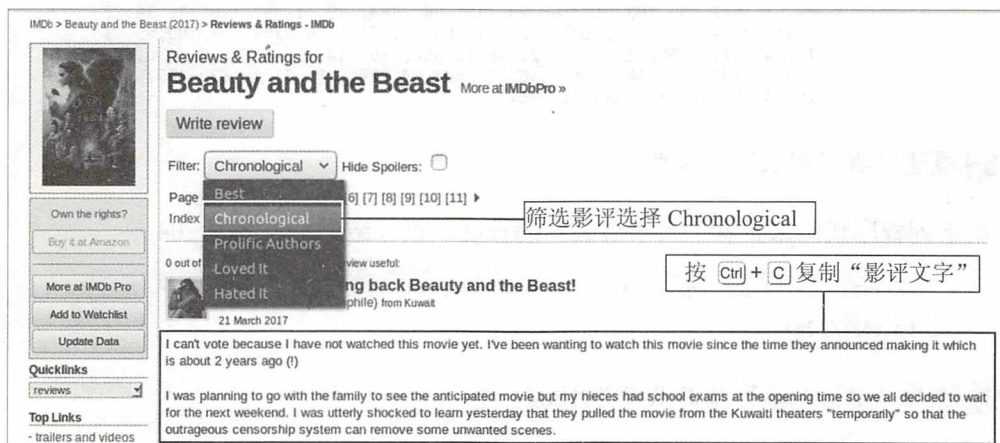


图 14-3

按 **Ctrl**+**V** 粘贴“影评文字”

```
In [40]: input_text='
I can't vote because I have not watched this movie yet. I've been wa
I was planning to go with the family to see the anticipated movie bu
The controversial gay "moment" according to my online research is ba
Based on the trailers, I think the movie is very promising and enter
Impatiently waiting for the Kuwaiti cinemas to bring back the movie.
'
```

步骤02 执行预测。

将上一步网页上的影评文字剪贴到下列程序代码中，创建 `input_text` 变量。

上面的 `input_text` 影评文字将送入之前所建立的多层感知器模型中进行预测，不过必须先进行数据预处理，步骤如下。

步骤03 将“影评文字”转换成“数字列表”。

以下程序代码使用 `token.texts_to_sequences` 将“影评文字”转换成“数字列表”，不过因为输入参数必须是字符串的列表，然而 `input_text` 是一个字符串，所以我们前后加上中括号 `[input_text]` 以便转换为字符串的列表。

```
In [41]: input_seq = token.texts_to_sequences([input_text])
```

步骤04 查看“数字列表”。

上一步的执行结果是“数字列表”的列表，因为只有一项数据，所以使用 `input_seq[0]` 查看第 0 项，也是唯一一项数据。

```
In [42]: print(input_seq[0])

[9, 187, 84, 9, 24, 20, 292, 10, 16, 242, 203, 73, 1780, 5, 102, 10
, 16, 233, 1, 54, 32, 227, 8, 59, 6, 40, 237, 149, 593, 9, 12, 5, 1
36, 15, 1, 219, 5, 63, 1, 16, 17, 57, 65, 391, 29, 1, 632, 54, 34,
71, 28, 867, 5, 853, 14, 1, 372, 9, 12, 1250, 5, 848, 11, 32, 1914,
1, 16, 35, 1, 34, 11, 1, 1502, 66, 45, 135, 1, 989, 557, 1790, 5, 5
7, 6, 1196, 46, 34, 9, 187, 165, 97, 279, 14, 28, 1, 194, 73, 166,
19, 2, 8, 12, 75, 191, 50, 2, 798, 7, 5, 512, 1, 16, 15, 360, 2, 15
61, 854, 56, 154, 8, 12, 7, 1, 2, 142, 180, 248, 87, 4, 144, 111, 1
84, 1, 577, 5, 102, 1, 16, 29, 11, 54, 444, 19, 1, 9, 100, 1, 16, 6
, 51, 2, 438, 2, 21, 187, 327, 1, 89, 5, 717, 10, 782, 5, 109, 5, 1
76, 4, 573, 250, 153, 15, 2, 1561, 854, 39, 14, 1, 4, 6, 9, 436, 80
, 524, 1921, 3, 16, 154, 256, 1, 581, 4, 145, 8, 7, 1, 82, 269, 106
1, 14, 1, 5, 717, 141, 1, 16]
```

步骤05 查看“数字列表”的长度。

使用下列程序代码来查看“数字列表”的长度，执行结果是长度为 204。

```
In [42]: len(input_seq[0])
Out[42]: 204
```

步骤06 截取“数字列表”使其长度为 100。

下列程序代码使用 `sequence.pad_sequences` 截取“数字列表”，使其长度为 100。



```
In [43]: pad_input_seq = sequence.pad_sequences(input_seq, maxlen=100)
```

步骤07 截长补短后查看“数字列表”的长度。

再使用下列程序代码来查看“数字列表”的长度。

```
In [44]: len(pad_input_seq[0])  
Out[44]: 100
```

从以上执行结果可以看到，经过 `sequence.pad_sequences` 处理后，长度变为 100。

步骤08 使用多层感知器模型进行预测。

下面的指令使用 `model.predict_classes` 传入参数 `pad_input_seq` 进行预测。

```
In [45]: predict_result=model.predict_classes(pad_input_seq)  
1/1 [=====] - 0s
```

步骤09 查看预测结果。

用下列指令来查看预测结果：

```
In [46]: predict_result  
Out[46]: array([[1]], dtype=int32)
```

可以看到，预测结果有前后有两个中括号`[[1]]`，所以这是一个二维数组，它只有一个元素。

步骤10 读取预测结果中的元素。

因为这是一个二维数组，所以我们使用 `predict_result[0][0]` 来读取其中的元素。

```
In [47]: predict_result[0][0]  
Out[47]: 1
```

从以上执行结果可以看到这一则影评文字的预测结果是 1，也就是正面的评价。

步骤11 执行预测。

最后，我们可以使用之前定义的 `SentimentDict` 字典将结果 1 转换为文字。

```
In [48]: SentimentDict[predict_result[0][0]]  
Out[48]: '正面的'
```

从以上执行结果可知，这是正面的评价。

14.10 预测《美女与野兽》的影评是正面或负面的

在本节中，我们将前面的命令全部整理成 `predict_review()` 函数，以便于预测其他《美女与野兽》的影评。

步骤01 创建 `predict_review()` 函数。

创建 `predict_review()` 函数如下：

```
In [49]: def predict_review(input_text):
         input_seq = token.texts_to_sequences([input_text])
         pad_input_seq = sequence.pad_sequences(input_seq, maxlen=100)
         predict_result=model.predict_classes(pad_input_seq)
         print(SentimentDict[predict_result[0]][0])
```

`predict_review()` 使用很简单，只需要传入参数 `input_text`（影评文字）就可以预测此影评是“正面”的或“负面”的。

步骤02 筛选 Hated It 影评。

我们可以在 IMDb 网站筛选 Hated It（讨厌的）影评，这些影评应该大部分是负面的评价，如图 14-4 所示。我们可以使用这些影评来验证模型的准确率。

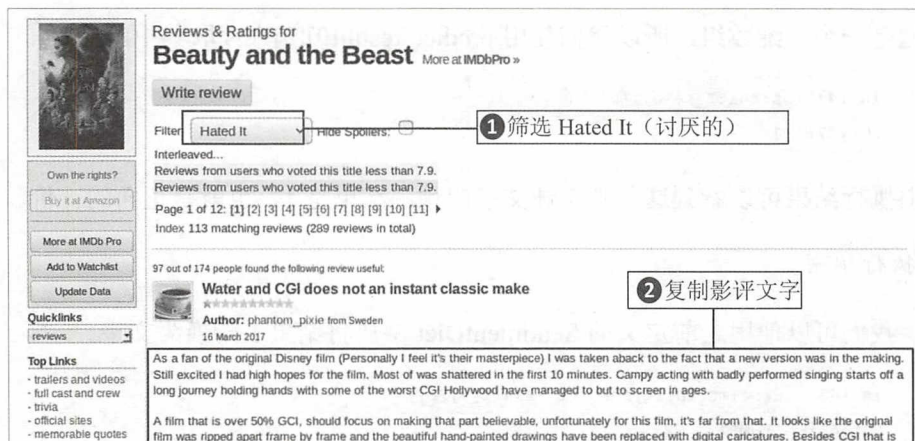


图 14-4

步骤03 执行预测。

接下来使用之前创建的 `predict_review()` 函数预测粘贴的“影评文字”。


```
In [69]: predict_review('')
As a fan of the original Disney film (Personally I feel it's their m
A film that is over 50% GCI, should focus on making that part believ
All and all, I might be bias from really loving Disney's first adapt
...)
```

1/1 [=====] - 0s
负面的

粘贴影评文字

从以上执行结果可以看到结果是负面的评价。其他读者可以自己筛选 Hated It 影评试试看，会发现大部分都是负面的评价。

步骤04 筛选 Loved It 影评。

我们可以在 IMDb 网站筛选 Loved It（喜欢的）影评，这些影评大部分是正面的评价，如图 14-5 所示。我们可以使用这些影评来验证模型的准确率。

The screenshot shows the IMDb page for 'Beauty and the Beast'. A filter 'Loved It' is selected, indicated by a red box and the annotation '1 影评筛选 Loved It (喜欢的)'. Below the filter, a review by 'INSANEUNO' is highlighted with a red box and the annotation '2 复制影评文字'. The review text is: 'Unpopular opinion: This is better than the original. Author: INSANEUNO. 19 March 2017. *** This review may contain spoilers *** The original Beauty and the Beast was my favorite cartoon as a kid but it did have major plot holes. Why had no one else ever seen the castle or knew where it was? Didn't anyone miss the people who were cursed? All of that gets an explanation when the enchantress places her curse in the beginning. Why did Belle and her Father move to a small town? Her mother died and the father thought it as best to leave. I love the new songs and added lyrics to the originals. I like the way the cgi beast looks (just the face is CGI). I think Emma Watson is a perfect Belle who is outspoken, fearless, and different. The set design is perfect for the era in France. I know a lot of people disagree but I found this remake with all its changes to be more enchanting, beautiful, and complete than the original 1991 movie. To each his own but I think everyone should see it for themselves.'

图 14-5

步骤05 执行预测。

接下来使用之前所创建的 predict_review()函数来预测粘贴的“影评文字”。

```
In [68]: predict_review('')
The original Beauty and the Beast was my favorite cartoon as a kid b
I know a lot of people disagree but I found this remake with all its
...)
```

1/1 [=====] - 0s
正面的

粘贴影评文字

从以上执行结果可以看到结果是正面的评价。读者可以自己剪切试试筛选 Loved It 影评，就会发现大部分都是正面的评价。

14.11

文字处理时使用较大的字典提取更多文字

之前模型预测的准确率是 0.80，我们希望能够再提高预测的准确率，方法如下。

- **建立字典的单词数：**原本是有 1000 个单词的字典，增加为建立有 3800 个单词的字典。
- **“数字列表”截长补短的长度：**原本“数字列表”的长度都是 100 个数字，现在改为 380 个数字。

这种方式就好像训练深度学习模型，多认识一些单词，并且增加读取影评文字的单词数，以增加准确率。以下可参考范例程序 Keras_Imdb_MLP_Large.ipynb。此程序代码大部分与 Keras_Imdb_MLP.ipynb 相同，此处只说明修改的部分。

步骤01 数据预处理。

数据预处理修改如下：

```
In [7]: #先读取所有文章建立字典，限制字典的单词数量为 nb_words=3800

In [8]: token = Tokenizer(num_words=3800)
token.fit_on_texts(train_text)  建立 3800 个单词的字典

In [9]: #将文字转为数字序列

In [10]: x_train_seq = token.texts_to_sequences(train_text)
x_test_seq = token.texts_to_sequences(test_text)

In [11]: #截长补短，让所有影评所产生的数字序列长度一样  “数字列表” 的长度为 380

In [12]: x_train = sequence.pad_sequences(x_train_seq, maxlen=380)
x_test = sequence.pad_sequences(x_test_seq, maxlen=380)
```

步骤02 建立模型。

建立模型修改如下：

```

In [14]: model = Sequential()

In [15]: model.add(Embedding(output_dim=32,
                             input_dim=3800,
                             input_length=380))
                             输入的维数修改为 3800
                             数字列表”的长度修改为380
model.add(Dropout(0.2))

In [16]: model.add(Flatten())

In [17]: model.add(Dense(units=256,
                          activation='relu' ))
model.add(Dropout(0.2))

In [18]: model.add(Dense(units=1,
                          activation='sigmoid' ))

In [19]: model.summary()

```

步骤03 修改 pad_sequences。

```

In [41]: pad_input_seq = sequence.pad_sequences(input_seq , maxlen=380)

```

“数字列表”的长度修改为380

步骤04 修改 predict_review 函数。

```

In [47]: def predict_review(input_text):
          input_seq = token.texts_to_sequences([input_text])
          pad_input_seq = sequence.pad_sequences(input_seq , maxlen=380)
          predict_result=model.predict_classes(pad_input_seq)
          print(SentimentDict[predict_result[0][0]])

```

“数字列表”的长度修改为380

步骤05 评估模型的准确率。

```

In [25]: scores = model.evaluate(x_test, y_test, verbose=1)
          scores[1]
          24992/25000 [=====>.] - ETA: 0s
Out[25]: 0.8534399999999998

```

经过修改模型后：把字典的单词数增加为 3800，并且“数字列表”的长度增加为 380。训练的时间比较长，但是这是值得的，准确率从 0.80 提高到 0.85。

14.12 RNN 模型介绍

接下来，我们将使用递归神经网络进行 IMDb 情感分析，并且训练模型、进行预测，最后产生预测结果（“正面评价”或“负面评价”）。

1. 为什么要使用 RNN 模型

之前我们介绍的 MNIST 数据集（识别数字图形）、Cifar 数据集（识别照片）图像并不会随着时间而改变，所以使用多层感知器或卷积神经网络都能达到不错的效果。

然而，人工智能所要解决的问题很多是顺序性的，例如自然语言处理（同一时间只能听到一个字，之前的语言会影响之后语言的含义）、视频图像处理（视频是一张张的照片，依照时间顺序所组成的）、气象观测数据（气象信息随着时间不断改变）和股票交易数据（股市开盘后，股价随着时间不断变动）。

以自然语言处理为例，当我们在听人说话时，因为同一个时间只能听一个字，所以会根据之前的时间点、所听到的话语来理解当前时间点这句话的意义。例如，“我家住上海市”“我在市政府上班”。因为前一句话已经说住在上海市，所以当我们理解后面那一句“我在市政府上班”时，通常是指上海市的市政府，不会是其他城市的市政府。

因为多层感知器（MLP）或卷积神经网络（CNN）都只能依照当前的状态进行识别，如果要处理时间序列的问题，就必须使用 RNN 与 LSTM 模型。

2. RNN 模型原理

RNN 模式的原理是将神经元的输出再接回神经元的输入。这样的设计使神经网络具备“记忆”功能，如图 14-6 所示。

说明如下：

- X 是神经网络的输入。
- O 是神经网络的输出。
- (U,V,W)都是神经网络的参数。
- S 是隐藏状态，代表着神经网络的“记忆”。

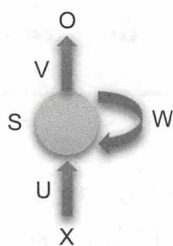


图 14-6

3. 以时间点展开 RNN 模型

为了让读者更容易理解，我们将之前的图以时间点展开，如图 14-7 所示。

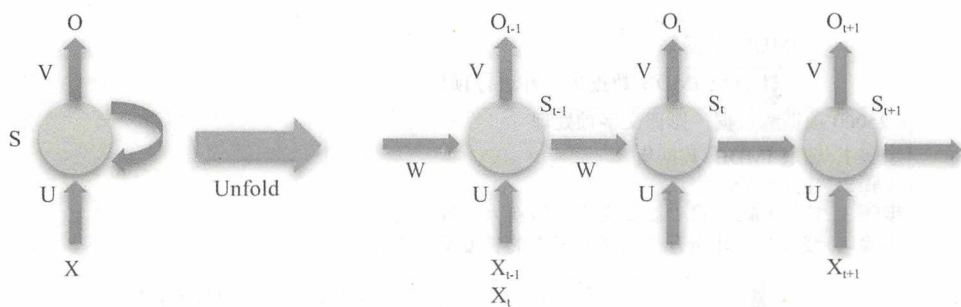


图 14-7

上图共有 3 个时间点，按照顺序是：“ $t-1$ ” “ t ” “ $t+1$ ”。

➤ 在 t 时间点

- X_t 是 t 时间点神经网络的输入。
- O_t 是 t 时间点神经网络的输出。
- (U, V, W) 都是神经网络的参数， W 参数是 $t-1$ 时间点的输出，但是作为 t 时间点的输入。
- S_t 是隐藏状态，代表着神经网络的“记忆”，是经过当前时间点的输入 X_t ，再加上前一个时间点的状态 S_{t-1} 和 U 、 W 的参数共同评估的结果，其公式如下：

$$S_t = f([U]X_t + [W]S_{t-1})$$

上面的 f 函数是非线性函数，例如 ReLU。

14.13 使用 Keras RNN 模型进行 IMDB 情感分析

在上一节，我们已经对 RNN 有了基本的了解，接下来将使用 RNN 模型进行 IMDB 情感分析，整理如图 14-8 所示。

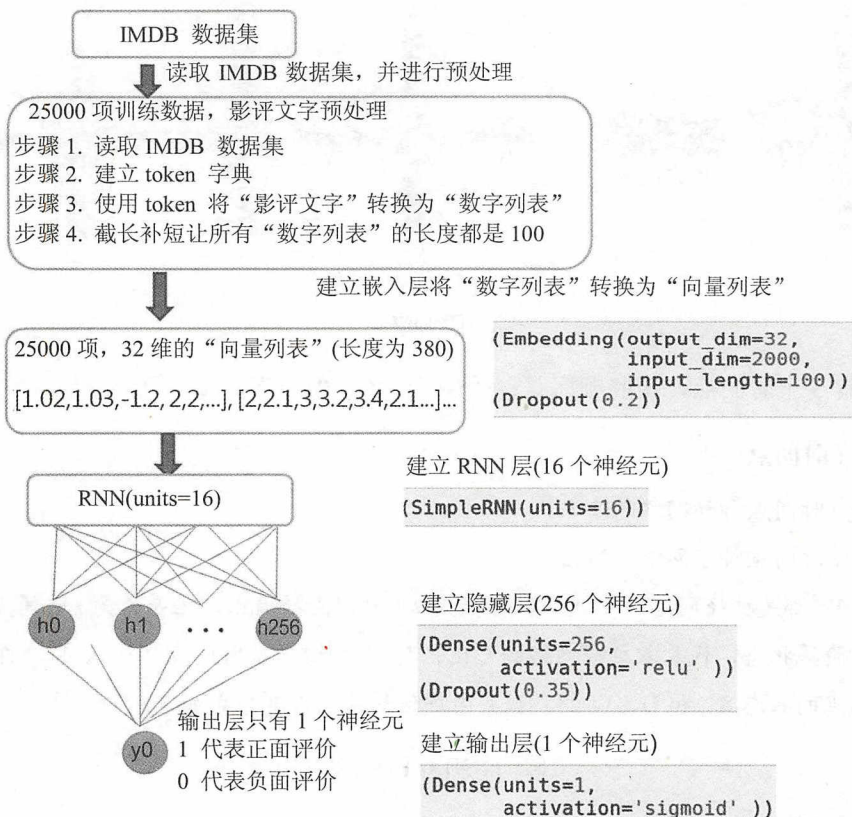


图 14-8

以下完整的程序代码可参考范例程序 Keras_Imdb_RNN.ipynb。此程序代码大部分与 Keras_Imdb_MLP_Large.ipynb 相同，以下只说明修改的部分。

步骤01 建立模型。

以下程序代码使用 SimpleRNN(unit=16)建立 16 个神经元的 RNN 层。

```
In [9]: from keras.models import Sequential
        from keras.layers.core import Dense, Dropout, Activation
        from keras.layers.embeddings import Embedding
        from keras.layers.recurrent import SimpleRNN

In [10]: model = Sequential()

In [11]: model.add(Embedding(output_dim=32,
                             input_dim=3800,
                             input_length=380))
        model.add(Dropout(0.35))

In [12]: model.add(SimpleRNN(units=16))  RNN 层

In [14]: model.add(Dense(units=256, activation='relu' ))
        model.add(Dropout(0.35))

In [15]: model.add(Dense(units=1, activation='sigmoid' ))
```


步骤02 查看模型的摘要。

我们可以使用下列指令来查看模型的摘要。

```
In [15]: model.summary()
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 380, 32)	121600
dropout_1 (Dropout)	(None, 380, 32)	0
simple_rnn_1 (SimpleRNN)	(None, 16)	784
dense_1 (Dense)	(None, 256)	4352
dropout_2 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 1)	257
Total params: 126,993.0		
Trainable params: 126,993.0		
Non-trainable params: 0.0		

嵌入层

RNN 层

隐藏层

输出层

步骤03 评估模型的准确率。

```
In [21]: scores = model.evaluate(x_test, y_test, verbose=1)
         scores[1]
```

```
24992/25000 [=====>.] - ETA: 0s - ETA: 39s
```

```
Out[21]: 0.84236
```

使用 RNN 模型准确率大约为 0.84。

14.14 LSTM 模型介绍

长短期记忆 (Long Short Term Memory, LSTM) 也是一种时间递归神经网络, 专门设计来解决 RNN 的长期依赖问题。

1. RNN 的长期依赖问题

之前介绍的 RNN 在训练时会有长期依赖的问题, 这是由于 RNN 模型在训练时会遇到梯度消失或爆炸的问题。训练时计算和反向传播, 梯度倾向于在每一时刻递增或递减, 经过一段时间后, 会发散到无穷大或收敛到零。

简单来说, 长期依赖的问题就是在每一个时间的间隔不断增大时, RNN 会丧失学习到连接到处处的信息的能力。

如图 14-9 所示, 随着时间点 t 不断递增, 由 0、1、2 一直到 $t-1$ 、 t 、 $t+1$ 。到了时间点的后期 t , 隐藏状态 (记忆) S_t 已经丧失了学习连接到远处的信息 X_0 的能力。

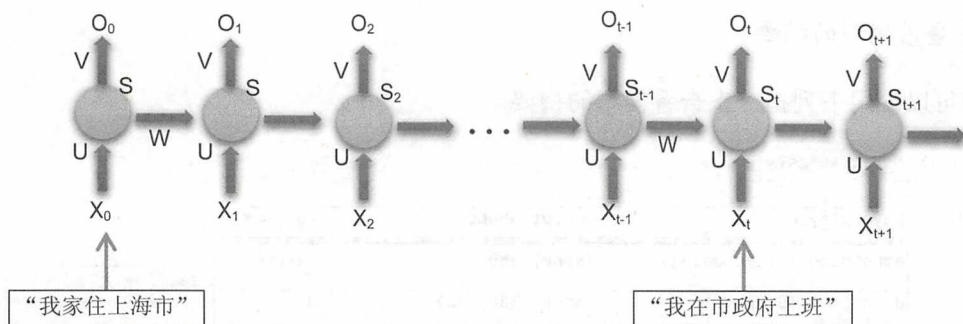


图 14-9

假设 X_0 输入“我家住上海市”，中间插了很多其他的句子，然后在 X_t 输入“我在市政府上班”，由于 X_0 与 X_t 相差很远，当 RNN 输入到 X_t 时，隐藏状态（记忆） S_t 已经丧失了学习连接远处的信息 X_0 的能力。当神经网络 X_t 输入“我在市政府上班”，由于已经忘记了 X_0 输入“我家住上海市”，因此神经网络无法理解我是在哪一个城市的市政府上班。

2. LSTM 介绍

简单地说，RNN 只有短期记忆，没有长期记忆，所以深度学习专家 Schmidhuber 提出了 LSTM 模型，专门设计来解决 RNN 的长期依赖问题，如图 14-10 所示。

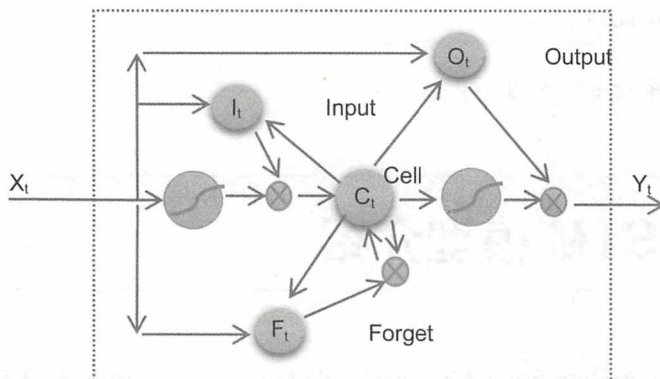


图 14-10

在 LSTM 神经网络中，每一个神经元相当于一个记忆细胞（cell），说明如下。

- X_t : 输入向量。
- Y_t : 输出向量。
- C_t : cell 是 LSTM 的记忆细胞的状态（cell state）。
- LSTM 通过一种名为“闸门”（Gate）的机制控制记忆细胞的状态，删减或增加其中的信息。
 - I_t : “输入闸门”（Input Gate）用于决定哪些信息要被增加到 cell。
 - F_t : “遗忘闸门”（Forget Gate）用于决定哪些信息要从 cell 删减。
 - O_t : “输出闸门”（Output Gate）用于决定哪些信息要从 cell 输出。



有了“闸门”机制，LSTM 就有了长期记忆。

14.15 使用 Keras LSTM 模型进行 IMDb 情感分析

在上一节，我们已经对 LSTM 有了基本的了解，接下来将使用 LSTM 模型进行 IMDb 情感分析。下面完整的程序代码可参考范例程序 `Keras_Imdb_LSTM.ipynb`。此程序代码大部分与 `Keras_Imdb_MLP_Large.ipynb` 相同，以下只说明修改的部分。

步骤01 建立模型。

下面的程序代码使用 `LSTM(32)` 建立 32 个神经元的 LSTM 层。

```
In [13]: from keras.models import Sequential
         from keras.layers.core import Dense, Dropout, Activation, Flatten
         from keras.layers.embeddings import Embedding
         from keras.layers.recurrent import LSTM

In [14]: model = Sequential()

In [15]: model.add(Embedding(output_dim=32,
                             input_dim=3800,
                             input_length=380))
         model.add(Dropout(0.2))

In [16]: model.add(LSTM(32))  LSTM 层

In [17]: model.add(Dense(units=256,
                           activation='relu' ))
         model.add(Dropout(0.2))

In [18]: model.add(Dense(units=1,
                           activation='sigmoid' ))

In [19]: model.summary()
```

步骤02 查看模型的摘要。

我们可以使用下列指令来查看模型的摘要。

In [19]: `model.summary()`

Layer (type)	Output Shape	Param #	
embedding_1 (Embedding)	(None, 380, 32)	121600	Embedding 层
dropout_1 (Dropout)	(None, 380, 32)	0	
lstm_1 (LSTM)	(None, 32)	8320	LSTM 层
dense_1 (Dense)	(None, 256)	8448	隐藏层
dropout_2 (Dropout)	(None, 256)	0	
dense_2 (Dense)	(None, 1)	257	输出层

Total params: 138,625.0
 Trainable params: 138,625.0
 Non-trainable params: 0.0

步骤03 评估模型的准确率。

```
In [37]: scores = model.evaluate(x_test, y_test, verbose=1)
          scores[1]

25000/25000 [=====] - 96s
Out[37]: 0.8619599999999995
```

使用 LSTM 模型准确率提升至约 0.86。

14.16 结论

在前面的章节中我们已经介绍了使用 Keras 进行手写数字识别、照片图像识别、预测泰坦尼克号上旅客的生存概率、影评文字情感分析。读者已经熟悉了深度学习模型的建立、训练和预测，后面的章节我们将介绍使用 TensorFlow 建立深度学习模型、训练模型和进行预测。

第15章

TensorFlow程序设计模式

在第 3 章中已经大致介绍了 TensorFlow 程序设计模式的概念，本章我们将以简单的程序来示范 TensorFlow 程序设计模式。而 TensorFlow 与 Keras 最大的差别是，对于 TensorFlow，我们必须自行设计张量（矩阵）运算，所以本章将介绍 TensorFlow 张量运算。

TensorFlow 程序设计模式的核心是“计算图”，可分为两部分：建立“计算图”与执行“计算图”，如图 15-1 所示。

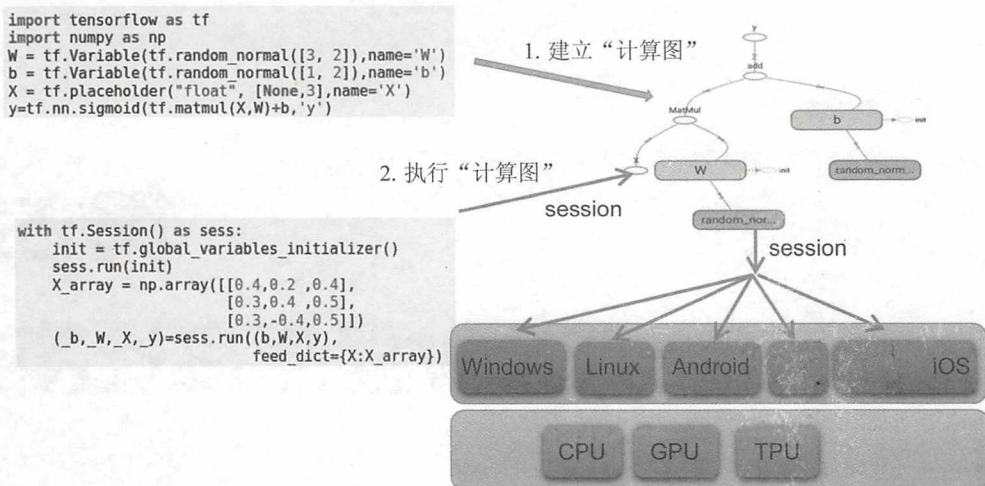


图 15-1

图 15-1 说明如下：

(1) 建立“计算图”

我们可以使用 TensorFlow 提供的模块建立“计算图”。TensorFlow 提供的模块非常大，可以设计张量运算流程，并且构建各种深度学习或机器学习模型。

(2) 执行“计算图”

建立“计算图”后，我们就可以建立 Session 执行“计算图”。在 TensorFlow 中，Session（原意是会话）的作用是在客户端和执行设备之间建立连接。有了这个连接，就可以将“计算图”在各种不同的设备中执行，后续任何与设备之间的数据传输都必须通过 Session 才能进行，执行“计算图”后会返回结果。

下面的程序代码可参考范例程序 TensorFlow_Basic.ipynb。范例程序的下载与安装可参考本书附录 A。

15.1 建立“计算图”

为了示范 TensorFlow 程序设计模式，我们将建立简单的“计算图”，只有一个常数与一个变量。建立完成后再执行此“计算图”，如图 15-2 所示。


```
import tensorflow as tf
ts_c = tf.constant(2,name='ts_c')
ts_x = tf.Variable(ts_c+5,name='ts_x')
```

1. 建立“计算图”

```
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    print('ts_c=',sess.run(ts_c))
    print('ts_x=',sess.run(ts_x))
```

2. 执行“计算图”

图 15-2

步骤01 导入 TensorFlow 模块。

```
In [1]: import tensorflow as tf
```

步骤02 建立 TensorFlow 常数。

建立 TensorFlow 常数命令如下：

```
In [2]: ts_c = tf.constant(2,name='ts_c')
```

以上程序代码使用 `tf.constant` 建立 TensorFlow 常数，需输入表 15-1 中的参数。

表 15-1 使用 `tf.constant` 建立 TensorFlow 常数需输入的参数

参数	说明
2	设置此常数值为 2
name= 'ts_c '	设置常数名称 <code>ts_c</code> ，此名称将会显示在计算图上

步骤03 查看 TensorFlow 常数。

查看 TensorFlow 常数指令如下：

```
In [3]: ts_c
Out[3]: <tf.Tensor 'ts_c:0' shape=() dtype=int32>
```

以上执行结果说明如下。

- **tf.Tensor**: 代表这是 TensorFlow 张量。
- **shape=()**: 代表这是零维的 tensor，也就是数值。
- **dtype=int32**: 代表此张量数据类型是 `int32`。

步骤04 建立 TensorFlow 变量。

建立 TensorFlow 变量的命令如下。

```
In [4]: ts_x = tf.Variable(ts_c+5,name='ts_x')
```

以上程序代码使用 `tf.Variable` 建立 TensorFlow 变量，需输入表 15-2 中的参数。

表 15-2 使用 `tf.Variable` 建立 TensorFlow 变量需输入的参数

参数	说明
<code>ts_c+5</code>	设置此变量值为常数值 <code>ts_c</code> 再加 5
<code>name= 'ts_x '</code>	设置变量名称 <code>ts_x</code> ，此名称将会显示在计算图上

➤ 查看 TensorFlow 变量

要查看 TensorFlow 变量，只要输入名称即可，指令如下。

```
In [5]: ts_x
Out[5]: <tensorflow.python.ops.variables.Variable at 0x7f649f9917b8>
```

从以上执行结果可以看到只显示这一个 TensorFlow 变量。这是因为 TensorFlow 变量必须要执行“计算图”之后，才能够看到结果。

15.2 执行“计算图”

建立“计算图”后，我们就可以执行“计算图”。只是执行之前必须先建立 Session（会话），在 TensorFlow 中 Session 代表在客户端和执行设备之间建立连接。有了这个连接，就可以在设备中执行“计算图”，后续任何与设备之间的沟通都必须通过这个 Session，并且可取得执行后的结果。

步骤01 建立 Session。

使用 `tf.Session()` 建立 Session 对象 `sess`。

```
In [5]: sess=tf.Session()
```

步骤02 执行 TensorFlow 来初始化变量。

必须使用下列指令初始化所有 TensorFlow global 变量。

```
In [6]: init = tf.global_variables_initializer()
sess.run(init)
```

步骤03 使用 `sess.run` 显示 TensorFlow 常数。

下面的程序代码使用 `sess.run` 执行 TensorFlow 的“计算图”，并且使用 `print` 显示 TensorFlow 常数的执行结果。

```
In [8]: print('ts_c=',sess.run(ts_c))
        ts_c= 2
```

从以上执行结果可以看到 ts_c 是常数，显示为 2。

步骤04 使用 sess.run 显示 TensorFlow 变量。

相同的方式也可以显示 TensorFlow 变量的执行结果。

```
In [9]: print('ts_x=',sess.run(ts_x))
        ts_x= 7
```

从以上执行结果可以看到 ts_x 是变量 7，也就是 ts_c 是 2，加 5 等于 7。

步骤05 使用 .eval() 方法显示 TensorFlow 常数。

另一个执行 TensorFlow “计算图” 的方法是使用 TensorFlow 对象的 eval() 方法，使用 eval() 方法必须传入 session 参数。

```
In [10]: print('ts_c=',ts_c.eval(session=sess))
          ts_c= 2
```

步骤06 使用 eval() 方法显示 TensorFlow 变量。

我们也可以使用 TensorFlow 对象的 eval() 方法来显示 TensorFlow 变量。

```
In [11]: print('ts_x=',ts_x.eval(session=sess))
          ts_x= 7
```

步骤07 关闭 TensorFlow Session。

当我们不需要再使用 Session 时，必须使用 sess.close() 关闭 Session。

```
In [13]: sess.close()
```

步骤08 将以上指令全部一起执行。

下面的程序代码是将前面步骤介绍的指令全部一起执行。

```
In [14]: import tensorflow as tf
          ts_c = tf.constant(2,name='ts_c')
          ts_x = tf.Variable(ts_c+5,name='ts_x')

          sess=tf.Session()
          init = tf.global_variables_initializer()
          sess.run(init)
          print('ts_c=',sess.run(ts_c))
          print('ts_x=',sess.run(ts_x))
          sess.close()

          ts_c= 2
          ts_x= 7
```

建立“计算图”

打开Session 开始执行“计算图”

关闭Session 结束执行“计算图”

步骤09 With 语句打开 Session 并且自动关闭。

在前面的步骤中，我们使用 `tf.Session()` 建立 Session，并且使用 `sess.close()` 关闭 Session。这种做法可能有以下问题：

- (1) 可能忘记关闭 Session。
- (2) 当程序执行中发生异常时，可能导致没有关闭 Session。

为了解决此问题，可以使用 With 语句：

with 关键词后面是建立的命令 `tf.Session()`，as 关键词后面是 Session 的变量 `sess`。

在 with 程序块中可使用 `sess` 变量与设备沟通，离开 with 程序块就自动关闭 Session。

```
In [9]: a = tf.constant(2,name='a')
        x = tf.Variable(a+5,name='x')
        with tf.Session() as sess:
            init = tf.global_variables_initializer()
            sess.run(init)
            print('a=',sess.run(a))
            print('x=',sess.run(x))
```

打开 session

with 程序块

```
a= 2
x= 7
```

15.3 TensorFlow placeholder

在前面的范例中，在建立“计算图”时，我们会设置 `ts_c` 常数值为 2，并且设置变量 `ts_x` 为 `ts_c` 加 5，这都是在建立“计算图”阶段就已经设置完成的。可是如果希望在执行“计算图”阶段才设置数值，那么就必须使用 placeholder。

步骤01 建立“计算图”。

以下建立两个 placeholder，分别是 width（宽）与 height（高），然后使用 `tf.multiply` 将 width 与 height 相乘，相乘后的结果是 area（面积）。

```
In [15]: width = tf.placeholder("int32")
        height = tf.placeholder("int32")
        area=tf.multiply(width,height)
```

执行后建立如图 15-3 所示的“计算图”。

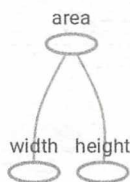


图 15-3

步骤02 执行“计算图”。

执行 `sess.run` 传入 `feed_dict` 参数 `{width: 6, height: 8}`。

```
In [16]: with tf.Session() as sess:
          init = tf.global_variables_initializer()
          sess.run(init)
          print('area=', sess.run(area, feed_dict={width: 6, height: 8}))

          area= 48
```

从以上执行结果可知，返回 `area=48`，也就是 `width` 是 6，`height` 是 8，相乘等于 48，如图 15-4 所示。

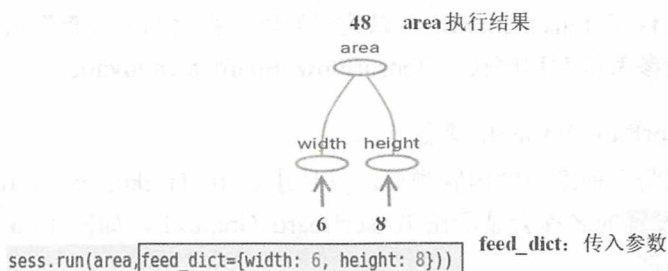


图 15-4

15.4 TensorFlow 数值运算方法介绍

在上一节中，我们使用 `tf.multiply()` 方法进行 TensorFlow 乘法运算。TensorFlow 提供了很多数值运算，可以参考下列 TensorFlow 说明文件：

https://www.tensorflow.org/api_docs/python/math_ops/

如果是双目数值运算，就输入两个参数：`x`、`y`。如果是单目数值运算（例如绝对值），就只传入参数 `x`。`name` 参数用于设置此运算名称。

表 15-3 所示为 TensorFlow 数值运算说明。

表 15-3 TensorFlow 数值运算说明

TensorFlow 数值运算	说明
<code>tf.add(x, y, name=None)</code>	加法
<code>tf.subtract(x, y, name=None)</code>	减法
<code>tf.multiply(x, y, name=None)</code>	乘法
<code>tf.divide(x, y, name=None)</code>	除法
<code>tf.mod(x, y, name=None)</code>	余数
<code>tf.sqrt(x, name=None)</code>	平方
<code>tf.abs(x, name=None)</code>	绝对值

注：文档中的数值运算方法很多，以上只列出常用的部分。

你也许会觉得只是简单的数值相乘，为什么要使用 `tf.multiply()` 方法？这是因为 TensorFlow 特别的程序设计模式必须以 TensorFlow 模块（例如 `tf.multiply()` 方法）来建立“计算图”，然后使用 `sess.run` 执行“计算图”，这样才能得到计算的结果。这样做的目的是让 TensorFlow 具备跨平台的能力。

15.5 TensorBoard

TensorFlow 提供了 TensorBoard，可以让我们以可视化的方式查看所建立的“计算图”。下面的程序代码请参考范例程序代码 `TensorFlow_Board_area.ipynb`。

1. 建立 TensorFlow Variable 变量

以下程序代码与之前章节的内容类似。只是建立 `tf.placeholder` 与 `tf.mul` 时加入了 `name` 参数，`name` 参数设置的名称会显示在 TensorBoard Graph 中，如图 15-5 所示。设置名称可以让“计算图”更易读。

```
import tensorflow as tf
width = tf.placeholder("int32", name='width')
height = tf.placeholder("int32", name='height')
area = tf.multiply(width, height, name='area')

with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    print('area=', sess.run(area, feed_dict={width: 6, height: 8}))
```

area= 48

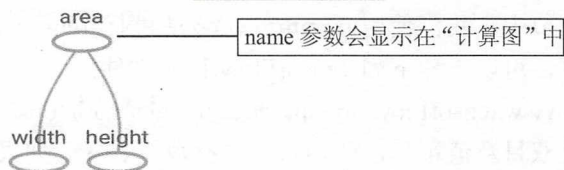


图 15-5

2. 建立 TensorFlow Variable 变量

下面的程序代码将要显示在 TensorBoard 的数据中写入 log 文件。

```
In [2]: tf.summary.merge_all()
        train_writer = tf.summary.FileWriter('log/area', sess.graph)
```

以上程序代码说明如下。

- **tf.summary.merge_all()**: 将所有要显示在 TensorBoard 的数据整合。
- **tf.summary.FileWriter()**: 将所有要显示在 TensorBoard 的数据写入 log 文件。log 文件会存储在当前程序执行目录下的 `log/area` 子目录中。

3. 在 Windows 中启动 TensorBoard

如果你使用的是 Windows 系统，就按照下列步骤启动 TensorBoard。再启动新的命令提示符程序，并且输入下列命令。

➤ 先确认 log 目录文件是否已经产生

在 Windows 的“命令提示符”程序中使用 `dir` 显示目录，以下 `C:\pythonwork\TensorFlow` 是程序的执行目录，如果读者的执行目录不同，修改为自己的执行目录即可。

```
dir c:\pythonwork\tensorFlow\log\area
```

➤ 启用 TensorFlow 的 Anaconda 虚拟环境

```
activate tensorflow
```

➤ 启动 TensorBoard

启动 TensorBoard 指令如下，需指定 log 文件目录，TensorBoard 会读取此目录，并显示在 TensorBoard 上。

```
tensorboard --logdir=c:\pythonwork\tensorFlow\log\area
```

执行后屏幕显示界面如图 15-6 所示。

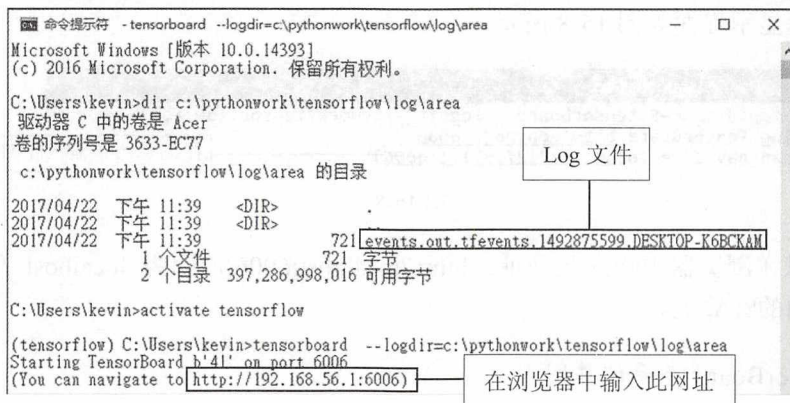


图 15-6

以上 `http://192.168.56.1:6006` 是笔者个人计算机的内部 IP，读者的 IP 可能不相同。读者也可以用浏览器输入此网址：`http://localhost:6006/`，因为 `localhost` 代表本机，就是读者当前正在使用的计算机。

4. 在 Linux Ubuntu 中启动 TensorBoard

如果读者使用的是 Linux Ubuntu 系统，就按照下列步骤启动 TensorBoard。

TensorFlow+Keras 深度学习人工智能实践应用

➤ 先确认 log 目录文件是否已经产生

可以使用下列指令来查看 log 目录，以下~/pywork/TensorFlow 是程序执行目录，如果读者的执行目录不同，就修改为自己的执行目录。

```
ll ~/pywork/tensorFlow/log/area
```

执行后屏幕显示界面如图 15-7 所示，从中可以看到 log 文件。

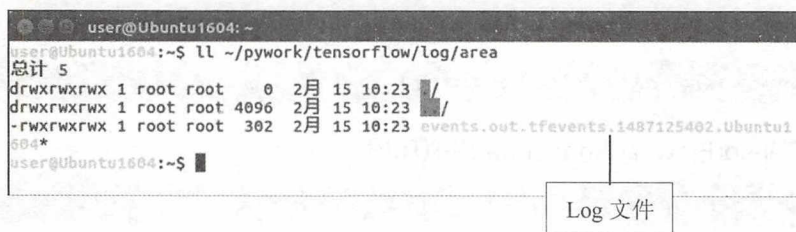


图 15-7

➤ 启动 TensorBoard

启动 TensorBoard 的指令如下，要先指定 log 文件目录，TensorBoard 会读取此目录，并显示在 TensorBoard 上。

```
tensorboard --logdir=~pywork/tensorFlow/log/area
```

执行后屏幕显示界面如图 15-8 所示。

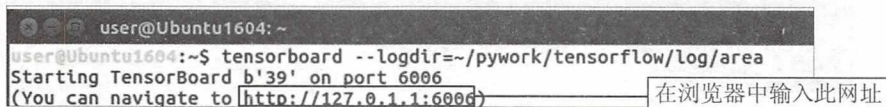


图 15-8

我们也可以在浏览器中输入此网址：http://localhost:6006/，因为 localhost 代表本机，就是我们当前使用的计算机。

5. 在 TensorBoard 查看计算图

启动 TensorBoard 之后，再启动浏览器，并输入网址：http://localhost:6006/。

输入网址后，就会出现 TensorBoard 界面，在菜单中选择 GRAPHS，之后就可以看到“计算图”，如图 15-9 所示。

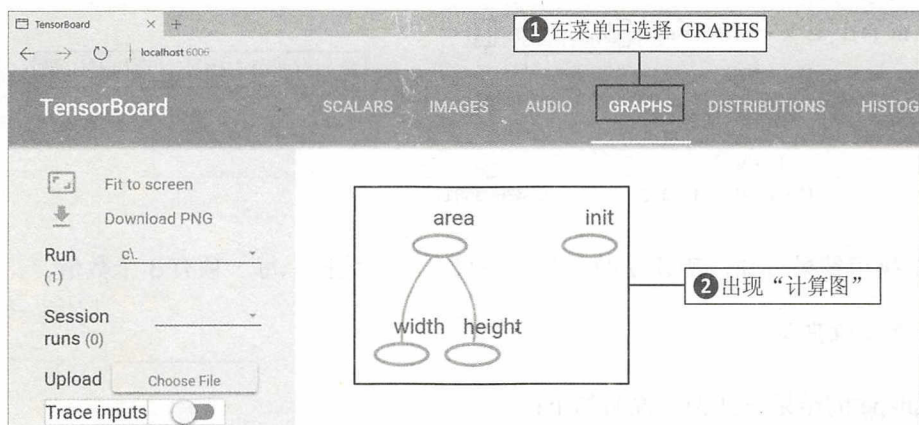


图 15-9

15.6 建立一维与二维张量

在前面的章节中，我们介绍了零维的张量，也就是标量（数值），接下来将介绍如何使用 TensorFlow 建立：一维的张量为向量，二维以上的张量为矩阵。

1. 建立一维张量（向量）

建立一维的张量（tensor）只需要使用 `tf.Variable()` 传入列表即可。

```
In [2]: ts_X = tf.Variable([0.4,0.2,0.4])

with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    X=sess.run(ts_X)
    print(X)

[ 0.40000001  0.2      0.40000001]
```

从以上执行结果可知，建立的是一维张量，共有 3 个数值。

2. 查看一维张量

可以使用 `shape` 查看数据的形状，因为一维张量共有 3 个数值，所以显示(3,.)。

```
In [3]: print(X.shape)

(3,)
```

3. 建立二维张量

建立二维的张量也是使用 `tf.Variable()` 传入二维的列表，所以我们传入列表的前后有两个中括号`[[0.4,0.2,0.4]]`，代表这是二维的列表。



```
In [4]: ts_X = tf.Variable([[0.4,0.2,0.4]])

with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    X=sess.run(ts_X)
    print(X)
```

前后有两个中括号，所以是二维

```
[[ 0.40000001  0.2          0.40000001]]
```

从以上执行结果可知，所建立的二维张量只有一项数据，每一项有 3 个数值。

4. 查看二维张量

查看 shape 的结果是(1,3)，说明如下：

```
In [5]: print('shape:',X.shape)
```

shape: (1, 3)

第一维，因为只有一项数据，所以是 1

第二维，每一项数据有 3 个数值，所以是 3

5. 再次建立二维张量

接下来，同样建立二维张量，共有 3 项数据，每一项数据有两个数值。

```
In [25]: W = tf.Variable([[-0.5,-0.2 ],
                           [-0.3, 0.4 ],
                           [-0.5, 0.2 ]])

with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    W_array=sess.run(W)
    print(W_array)
```

```
[[ -0.5      -0.2      ]
 [ -0.30000001  0.40000001]
 [ -0.5        0.2      ]]
```

3 项数据，每一项数据有两个数值

6. 查看新的二维张量

查看 shape 结果是(3,2)，说明如下：

```
In [23]: print(W_array.shape)
```

(3, 2)

第二维，因为只有 3 项数据，所以是 3

第二维，每一项数据有两个数值，所以是 2

15.7 矩阵基本运算

接下来将介绍矩阵基本运算：乘法与加法。

1. 矩阵乘法

TensorFlow 提供 `tf.matmul()` 方法，可以进行矩阵乘法。当进行矩阵乘法时，两个矩阵的维数必须相同。范例如下：

- 建立“计算图” 先建立 TensorFlow 变量 `X` 与 `W`，都是二维张量（矩阵），然后使用 `tf.matmul` 进行矩阵的相乘。
- 执行“计算图” 使用 `sess.run(XW)` 执行计算图，并用 `print` 显示结果。

```
In [24]: X = tf.Variable([[1.,1.,1.]])
        W = tf.Variable([[-0.5,-0.2 ],
                        [-0.3, 0.4 ],
                        [-0.5, 0.2 ]])
        XW = tf.matmul(X,W )

        with tf.Session() as sess:
            init = tf.global_variables_initializer()
            sess.run(init)
            print(sess.run(XW ))

        [[-1.29999995  0.40000001]]
```

建立“计算图”

矩阵的运算公式为 $XW=X \times W$ ，计算方式如下：

$$\begin{aligned}
 & \begin{bmatrix} 1.0 & 1.0 & 1.0 \end{bmatrix} \times \begin{bmatrix} -0.5 & -0.2 \\ -0.3 & 0.4 \\ -0.5 & 0.2 \end{bmatrix} \\
 &= [(1 \times -0.5 + 1 \times -0.3 + 1 \times -0.5) \quad (1 \times -0.2 + 1 \times 0.4 + 1 \times 0.2)] \\
 &= [-1.3 \quad 0.4]
 \end{aligned}$$

大家也许会觉得奇怪，为什么程序执行结果是 `[[-1.29999995 0.40000001]]`，这是因为矩阵运算是浮点运算，所以是近似值，与真实的计算结果会有误差。

2. 矩阵加法

TensorFlow 也可以进行矩阵加法，方法很简单，只需要使用加号即可。

```
In [25]: b = tf.Variable([[ 0.1,0.2]])
        XW = tf.Variable([[-1.3,0.4]])

        Sum = XW+b
        with tf.Session() as sess:
            init = tf.global_variables_initializer()
            sess.run(init)
            print('Sum:')
            print(sess.run(Sum ))

        Sum:
        [[-1.19999993  0.60000002]]
```

矩阵的运算公式为 $\text{Sum}=XW+b$ ，计算方式如下：

$$\begin{aligned} &[-1.3 \quad 0.4] + [0.1 \quad 0.2] \\ &= [-1.2 \quad 0.6] \end{aligned}$$

以上运算结果是 $\begin{bmatrix} -1.19999993 & 0.60000002 \end{bmatrix}$ ，因为是浮点运算，所以是近似值。

3. 矩阵乘法与加法

之前的步骤分别介绍了矩阵乘法与加法，接下来将乘法与加法一起运用。

```
In [26]: X = tf.Variable([[1.,1.,1.]])
        W = tf.Variable([[-0.5,-0.2 ],
                        [-0.3, 0.4 ],
                        [-0.5, 0.2 ]])
        b = tf.Variable([[0.1,0.2]])
        XWb =tf.matmul(X,W)+b

        with tf.Session() as sess:
            init = tf.global_variables_initializer()
            sess.run(init)
            print('XWb:')
            print(sess.run(XWb ))

XWb:
[[ -1.19999993  0.60000002]]
```

建立“计算图”

矩阵的运算公式为 $XWB=X \times W+b$ ，计算方式如下：

$$\begin{aligned} &[1.0 \quad 1.0 \quad 1.0] \times \begin{bmatrix} -0.5 & -0.2 \\ -0.3 & 0.4 \\ -0.5 & 0.2 \end{bmatrix} + [0.1 \quad 0.2] \\ &= [(1 \times -0.5 + 1 \times -0.3 + 1 \times -0.5) \quad (1 \times -0.2 + 1 \times 0.4 + 1 \times 0.2)] + [0.1 \quad 0.2] \\ &= [-1.3 \quad 0.4] + [0.1 \quad 0.2] \\ &= [-1.2 \quad 0.6] \end{aligned}$$

以上运算结果为 $\begin{bmatrix} -1.19999993 & 0.60000002 \end{bmatrix}$ ，因为是浮点运算，所以是近似值。

15.8 结论

本章介绍了 TensorFlow 的程序设计模式，并介绍了如何使用 TensorFlow 基本的张量运算，有了这些基础知识，下一章将介绍以 TensorFlow 张量（矩阵）运算来模拟类神经网络的运行。

第16章

以TensorFlow张量运算仿真神经网络的运行

在第2章中，我们介绍了以矩阵数学公式来仿真类神经网络的运行。在本章中，我们将以TensorFlow张量（矩阵）运算来仿真类神经网络的运行。

下面的程序代码可参考范例程序 TensorFlow_Tensor_neural.ipynb。有关范例程序下载与安装的细节可参考本书附录 A 中的“本书范例程序的下载与安装说明”。

16.1 以矩阵运算仿真神经网络

1. 以矩阵运算仿真神经网络的信息传导

可参考第 2 章以矩阵运算仿真神经网络的信息传导，如图 16-1 所示。

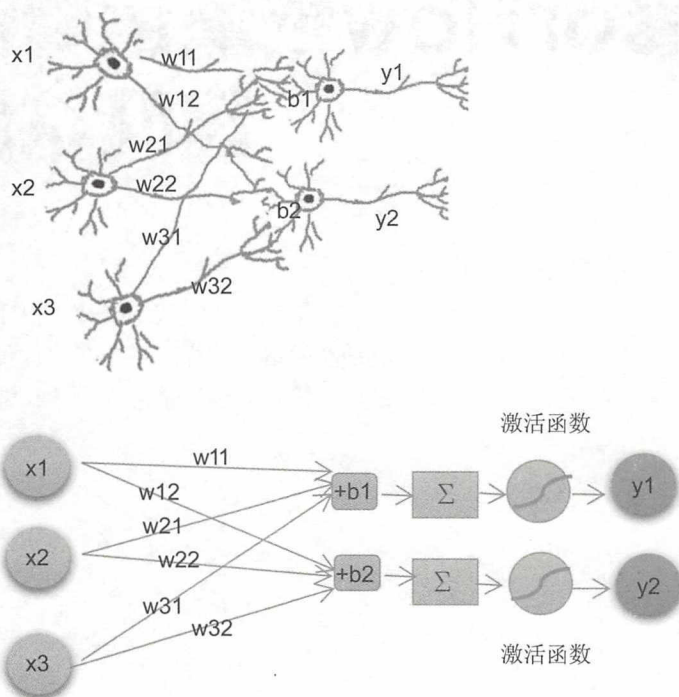


图 16-1

➤ 以数学公式模拟输出与接收神经元的工作方式：

$$y1 = \text{activation function} (x1 \times w11 + x2 \times w21 + x3 \times w31 + b1)$$

$$y2 = \text{activation function} (x1 \times w12 + x2 \times w22 + x3 \times w32 + b2)$$

➤ 以上两个数学公式可以整合成一个矩阵运算公式：

$$[y1 \ y2] = \text{activation} \left([x1 \ x2 \ x3] \times \begin{bmatrix} w11 & w12 \\ w21 & w22 \\ w31 & w32 \end{bmatrix} + [b1 \ b2] \right)$$



➤ 另一种形式的矩阵公式表示如下:

$$y = \text{activation}(X \times W + b)$$

➤ 矩阵公式以中文表示如下:

输出 = 激活函数 (输入 × 权重 + 偏差)

说明见表 16-1。

表 16-1 以矩阵运算模拟神经网络参数说明

项目	说明
输入 X	X 仿真输入神经元接收外界传送信息, 共有 3 个输入神经元: x_1 、 x_2 、 x_3
接收 y	y 模拟接收神经元, 共有两个输出神经元: y_1 、 y_2
权重 W	权重模拟神经元的轴突, 连接输入与接收神经元, 负责传送信息。因为要完全连接输入与接收神经元, 所以共需(输入 3)×(接收 2)=6 个轴突。 w_{11} 、 w_{21} 、 w_{31} 负责从 x_1 、 x_2 、 x_3 传送信息给 y_1 。 w_{12} 、 w_{22} 、 w_{32} 负责从 x_1 、 x_2 、 x_3 传送信息给 y_2
偏差值 b	偏差值 b 仿真突触的结构, 代表接收神经元容易被活化的程度, 偏差值越高, 越容易被活化并传递信息。因为接收神经元有两个, 所以也有两个偏差值: b_1 、 b_2
激活函数	激活函数仿真神经传导的运行方式, 例如, 当接收神经元 y_1 接收刺激的总和: $(x_1 \times w_{11} + x_2 \times w_{21} + x_3 \times w_{31} + b_1)$ 经过激活函数的运算大于临界值时, 就会传递到下一个神经元

2. TensorFlow 张量运算仿真神经网络

接下来, 我们将以 TensorFlow 张量运算模拟以上公式。

```
In [2]: X = tf.Variable([[0.4,0.2,0.4]])
        W = tf.Variable([[-0.5,-0.2 ],
                        [-0.3, 0.4 ],
                        [-0.5, 0.2 ]])
        b = tf.Variable([[0.1,0.2]])
        XWb =tf.matmul(X,W)+b
        y=tf.nn.relu(tf.matmul(X,W)+b)
        with tf.Session() as sess:
            init = tf.global_variables_initializer()
            sess.run(init)
            print('XWb:');print(sess.run(XWb ))
            print('y:');print(sess.run(y ))
```

XWb:	[[[-0.35999998 0.28]]	X×W+b 的计算结果
y:	[[0. 0.28]]	y = relu(X×W+b) 的计算结果

以上运算结果为 $y = \begin{bmatrix} 0 & 0.28 \end{bmatrix}$, 矩阵的运算方式如下:

$$\begin{aligned}
 & \text{relu} \left(\begin{bmatrix} 0.4 & 0.2 & 0.4 \end{bmatrix} \times \begin{bmatrix} -0.5 & -0.2 \\ -0.3 & 0.4 \\ -0.5 & 0.2 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.2 \end{bmatrix} \right) \\
 &= \text{relu}((0.4 \times (-0.5) + 0.2 \times -0.3 + 0.4 \times (-0.5)) \quad (0.4 \times (-0.2) + 0.2 \times 0.4 + 0.4 \times 0.2) + \begin{bmatrix} 0.1 & 0.2 \end{bmatrix}) \\
 &= \text{relu}(\begin{bmatrix} -0.46 & 0.08 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.2 \end{bmatrix}) \\
 &= \text{relu}(\begin{bmatrix} -0.36 & 0.28 \end{bmatrix}) \\
 &= \begin{bmatrix} 0 & 0.28 \end{bmatrix}
 \end{aligned}$$

在 TensorFlow 使用 `tf.nn.relu` 来引用 ReLU 激活函数。ReLU 的特色是，如果小于 0 转换后是 0，如果大于 0 输出等于输入。所以第 1 个元素 -0.36 经过 ReLU 转换后是 0，第 2 个元素 0.28 经过 ReLU 转换后仍然是 0.28。

3. 矩阵表达式加入 Sigmoid 激活函数

之前使用的是 ReLU 激活函数，我们也可以使用 Sigmoid 激活函数，程序代码如下：

```
In [3]: X = tf.Variable([[0.4,0.2,0.4]])
        W = tf.Variable([[-0.5,-0.2 ],
                        [-0.3, 0.4 ],
                        [-0.5, 0.2 ]])
        b = tf.Variable([[0.1,0.2]])
        XWb=tf.matmul(X,W)+b
        y=tf.nn.sigmoid(tf.matmul(X,W)+b)

        with tf.Session() as sess:
            init = tf.global_variables_initializer()
            sess.run(init)
            print('XWb:')
            print(sess.run(XWb))
            print('y:')
            print(sess.run(y))
```

XWb:		X×W+b 的计算结果
[[[-0.35999998 0.28]]		
y:		sigmoid(X×W+b) 的计算结果
[[[0.41095957 0.56954622]]		

在 TensorFlow 使用 `tf.nn.sigmoid` 来引用 Sigmoid 激活函数，计算方式如下：

$$\begin{aligned}
 & \text{sigmoid} \left([0.4 \ 0.2 \ 0.4] \times \begin{bmatrix} -0.5 & -0.2 \\ -0.3 & 0.4 \\ -0.5 & 0.2 \end{bmatrix} + [0.1 \ 0.2] \right) \\
 &= \text{sigmoid}((0.4 \times (-0.5) + 0.2 \times -0.3 + 0.4 \times (-0.5)) \quad (0.4 \times (-0.2) + 0.2 \times 0.4 + 0.4 \times 0.2) + [0.1 \ 0.2]) \\
 &= \text{sigmoid}([-0.46 \ 0.08] + [0.1 \ 0.2]) \\
 &= \text{sigmoid}([-0.36 \ 0.28]) \\
 &= [0.41095957 \ 0.56954622]
 \end{aligned}$$

4. 以正态分布的随机数生成权重与偏差的初始值

参考第 2 章，对于深度学习模型，我们会以反向传播算法进行训练，训练前必须先“建立模型”，建立多层感知模型必须以随机数初始化模型的权重与偏差。TensorFlow 提供 `tf.random_normal` 可以用来产生正态分布的随机数的矩阵。



```
In [4]: W = tf.Variable(tf.random_normal([3, 2]))
b = tf.Variable(tf.random_normal([1, 2]))
X = tf.Variable([0.4, 0.2, 0.4])
y=tf.nn.relu(tf.matmul(X,W)+b)
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    print('b:');print(sess.run(b))
    print('W:');print(sess.run(W))
    print('y:');print(sess.run(y))
```

```
b:
[[ 1.0690937  1.2734251]]
W:
[[ 1.429407  1.34082055]
 [ 0.07711758  0.96682191]
 [-3.24948716  0.37241682]]
y:
[[ 0.35648513  2.15208435]]
```

tf.random_normal 产生随机数来初始化权重与偏差

y = relu(X×W+b) 的执行结果

5. 执行一次 sess.run 可以取得 3 个 TensorFlow 变量

前面我们执行了 sess.run(b)、sess.run(W)、sess.run(y)，其实可以用另一种写法，只执行一次 sess.run 就可以取得 3 个 TensorFlow 变量，(b,W,y)=sess.run((b,W,y))。

```
In [5]: W = tf.Variable(tf.random_normal([3, 2]))
b = tf.Variable(tf.random_normal([1, 2]))
X = tf.Variable([0.4, 0.2, 0.4])
y=tf.nn.relu(tf.matmul(X,W)+b)
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    (b, W, y)=sess.run((b,W,y))
    print('b:')
    print(b)
    print('W:')
    print(W)
    print('y:')
    print(y)
```

执行一次 sess.run 即可

```
b:
[[ 0.52283692  2.35796475]]
W:
[[-0.02047499  0.43140945]
 [ 0.48463389 -1.13369405]
 [ 0.82369471  0.14685473]]
y:
[[ 0.9410516  2.36253166]]
```

因为随机数生成 W 与 b，所以每次执行结果都会不同

6. 正态分布的随机数 tf.random_normal

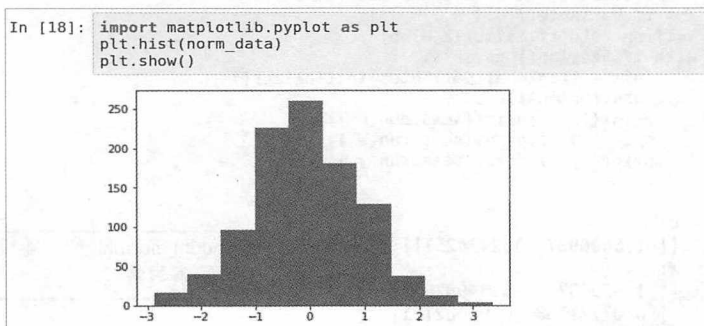
大家也许会好奇，什么是正态分布的随机数，我们将写个简单的程序代码来说明。

➤ 先使用 tf.random_normal 产生正态分布的随机数列表

```
In [17]: ts_norm = tf.random_normal([1000])
with tf.Session() as session:
    norm_data=ts_norm.eval()
    print(norm_data[:5])
```

```
[ 0.17875326  1.96279573 -0.8681004 -0.01844308 -0.72754133]
```

➤ 以 plt.hist 显示正态分布图形



16.2 以 placeholder 传入 X 值

x_1 、 x_2 、 x_3 是神经网络的输入，所以可能是任何数值，在实际运用时，会以 placeholder 传入神经网络进行运算。如图 16-2 所示，我们将 x_1 、 x_2 、 x_3 改为 place holder，后续以 `sess.run()` 执行“数据流程图”时，可以使用 `feed_dict` 传入数组进行运算。

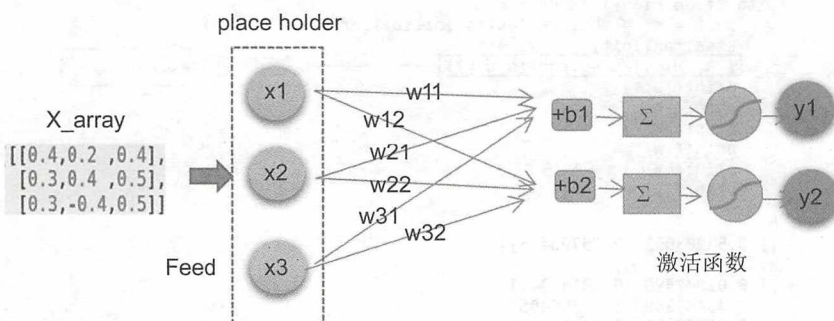


图 16-2

1. 以 placeholder 传入 1×3 的二维数组

以下程序代码使用 placeholder 传入二维数组 $X = [[0.4, 0.2, 0.4]]$ 。

```
In [6]: W = tf.Variable(tf.random_normal([3, 2]))
b = tf.Variable(tf.random_normal([1, 2]))
X = tf.placeholder("float", [None, 3])
y = tf.nn.relu(tf.matmul(X, W) + b)
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    X_array = np.array([[0.4, 0.2, 0.4]])
    (_b, W, X, y) = sess.run((b, W, X, y), feed_dict={X: X_array})
    print('b:'); print(_b)
    print('W:'); print(W)
    print('X:'); print(X)
    print('y:'); print(y)
```

① 建立“计算图”时，定义placeholder X

② 建立X_array

③ 执行“计算图”时，placeholder X 以 feed_dict 传入 X_array



```
b:
[[ 0.38106519 -0.73915875]]
W:
[[-0.28878763  0.9731341 ]
 [-0.16963108  0.75645941]
 [ 0.74396002 -0.07577042]]
X:
[[ 0.40000001  0.2          0.40000001]]
y:
[[ 0.52920794  0.          ]]
```

以上传入的 X 是 1×3 的二维数组 $[[0.4, 0.2, 0.4]]$ ，输出的 y 是 1×2 的二维数组 $[[0.52920794, 0.]]$ ，以上程序代码的说明如下：

➤ 先定义 placeholder

```
X = tf.placeholder("Float", [None, 3])
```

`tf.placeholder` 共有两个参数：

- 第 1 个参数设置为 "Float"，是 placeholder 的数据类型。
- 第 2 个参数设置为 `[None, 3]`，placeholder 矩阵的形状。第一维设置为 `None`。因为传入的 X 项数不限。第二维是每一项的数字个数，每一项有 3 个数字，所以设置为 3。

➤ 建立 X_array

```
X_array = np.array([[0.4, 0.2, 0.4]])
```

使用 `np.array` 建立 `X_array`。

➤ 执行 sess.run

```
(_b, _W, _X, _y) = sess.run((b, W, X, y), feed_dict={X: X_array})
```

`sess.run` 执行“数据流程图”，以 `feed_dict` 传入 X ：为 `X_array`，执行结果返回 `_b`、`_W`、`_X`、`_y` 变量。

2. 以 placeholder 传入 3×3 的二维数组

之前 placeholder 的传入是 1×3 的二维数组，接下来将传入 3×3 的二维数组进行计算。

```
In [10]: W = tf.Variable(tf.random_normal([3, 2]))
b = tf.Variable(tf.random_normal([1, 2]))
X = tf.placeholder("float", [None, 3])
y = tf.nn.sigmoid(tf.matmul(X, W) + b)
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    X_array = np.array([[0.4, 0.2, 0.4],
                        [0.3, 0.4, 0.5],
                        [0.3, -0.4, 0.5]])
    (_b, _W, _X, _y) = sess.run((b, W, X, y), feed_dict={X: X_array})
    print('b:'); print(_b)
    print('W:'); print(_W)
    print('X:'); print(_X)
    print('y:'); print(_y)
```

传入 3×3 的数组 `X_array`

```

b:
[[ 0.69074857 -0.54822081]]
W:
[[ 0.87199128  2.05090261]
 [-0.56881934 -1.05732417]
 [ 0.63020885 -0.18801321]]
X:
[[ 0.40000001  0.2      0.40000001]
 [ 0.30000001  0.40000001  0.5      ]
 [ 0.30000001 -0.40000001  0.5      ]]
y:
[[ 0.76456374  0.49636757]
 [ 0.7388351   0.38938782]
 [ 0.81682426  0.59771973]]

```

输出 3×2 的数组y

16.3 创建 layer 函数以矩阵运算仿真神经网络

前面的章节已经示范了以矩阵运算仿真神经网络，后续的章节我们将以相同的方式来建立类神经网络多层感知器，为了方便后续使用，我们将创建 layer 函数。

1. layer 函数

我们将创建下面的 layer 函数，其功能是建立两层神经网络。

```

In [21]: def layer(output_dim,input_dim,inputs, activation=None):
          W = tf.Variable(tf.random_normal([input_dim, output_dim]))
          b = tf.Variable(tf.random_normal([1, output_dim]))
          XWb = tf.matmul(inputs, W) + b
          if activation is None:
              outputs = XWb
          else:
              outputs = activation(XWb)
          return outputs

```

以上程序代码的详细说明如下：

➤ 定义 layer 函数参数

```
def layer(output_dim,input_dim,inputs, activation=None):
```

- output_dim: 输出的神经元数量。
- input_dim: 输入的神经元数量。
- input: 输入的二维数组 placeholder。
- activation: 传入激活函数，默认是 None。

➤ 以正态分布的随机数建立并且初始化 W (权重)

```
W = tf.Variable(tf.random_normal([input_dim, output_dim]))
```

以 tf.random_normal 函数传入 [input_dim,output_dim] 参数就可以产生维数是 (input_dim,output_dim) 的正态分布的随机数矩阵。

➤ 以正态分布的随机数建立 b (偏差)

```
b = tf.Variable(tf.random_normal([1, output_dim]))
```

以 `tf.random_normal` 函数传入 `[1,output_dim]` 参数就可以产生维数是 `(1,output_dim)` 的正态分布的随机数矩阵。

➤ 建立矩阵表达式 $XWb=(inputs \times W)+b$

```
XWb = tf.matmul(inputs, W) + b
```

➤ 设置 activation 激活函数

```
if activation is None: outputs = XWb
else:
    outputs = activation(XWb)
```

如果输入的参数 `activation` 是 `None`, 就不要使用激活函数, 如果传入激活函数, 就会使用传入的激活函数进行转换。

➤ 返回已建立的神经网络层

```
return outputs
```

2. 使用 layer 函数建立 3 层类神经网络

接下来, 我们将以 `layer` 函数建立 3 层类神经网络, 输入层有 4 个神经元, 隐藏层有 3 个神经元, 输出层有 2 个神经元, 如图 16-3 所示。

```
x = tf.placeholder("float",
                    [None, 4])
```

```
h=layer(output_dim=3,
        input_dim=4,
        inputs=x,
        activation=tf.nn.relu)
```

```
y=layer(output_dim=2,
        input_dim=3,
        inputs=h)
```

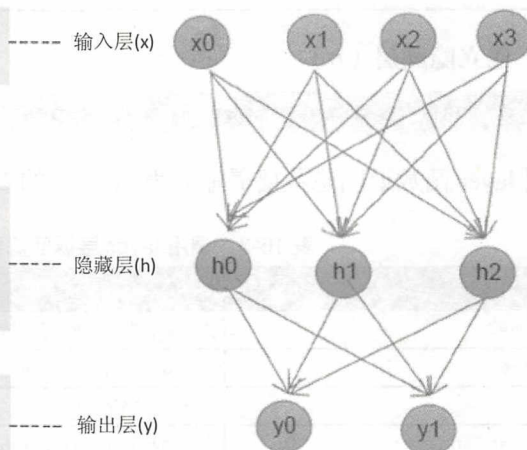


图 16-3

建立的程序代码如下:


```

In [13]: X = tf.placeholder("float", [None,4])
          h=layer(output_dim=3,input_dim=4,inputs=X,
                  activation=tf.nn.relu)
          y=layer(output_dim=2,input_dim=3,inputs=h)
          with tf.Session() as sess:
              init = tf.global_variables_initializer()
              sess.run(init)
              X_array = np.array([[0.4,0.2,0.4,0.5]])
              (layer_X,layer_h,layer_y)= \
                  sess.run((X,h,y),feed_dict={X:X_array})
              print('input Layer X:');print(layer_X)
              print('hidden Layer h:');print(layer_h)
              print('output Layer y:');print(layer_y)

input Layer X:
[[ 0.40000001  0.2          0.40000001  0.5          ]]
hidden Layer h:
[[ 0.  0.  0.]]
output Layer y:
[[-2.9684186  1.22442055]]

```

从以上执行结果可知，输入的 X 是 1×4 的张量，隐藏层 (h) 是 1×3 的张量，输出层 (y) 是 1×2 的张量，程序代码说明如下：

➤ 建立输入层 (X)

```
X = tf.placeholder("Float", [None,4])
```

建立输入层使用 `tf.placeholder` 方法，设置表 16-2 中的参数。

表 16-2 使用 `tf.placeholder` 方法建立输入层需设置的参数

参数	说明
"Float"	数据类型是 Float
[None, 4])	<ul style="list-style-type: none"> 第一维：设置为 None，因为项数不固定，所以设置为 None 第二维：设置为 4，因为输入神经元是 4

➤ 建立隐藏层 (h)

```
h=layer(output_dim=3,input_dim=4,inputs=X,activation=tf.nn.relu)
```

调用 `layer` 函数返回隐藏层需输入表 16-3 中的参数。

表 16-3 调用 `layer` 层返回隐藏层需输入的参数

参数	说明
output_dim=3	建立隐藏层神经元个数 3
input_dim=4	X (输入层)的神经元个数 4
inputs=X	传入 X (输入层)
Activation=tf.nn.relu	定义激活函数 <code>tf.nn.relu</code>

➤ 建立输出层 (y)

```
y=layer(output_dim=2,input_dim=3,inputs=h)
```

调用 `layer` 函数返回输出层需输入表 16-4 中的参数。



表 16-4 调用 layer 层返回输出层需要输入的参数

参数	说明
output_dim=2	建立输出层神经元个数 2
input_dim=3	h（隐藏层）的神经元个数 3
inputs= h1	传入 h（隐藏层）

16.4 建立 layer_debug 函数显示权重与偏差

之前 layer 函数只返回了 output，并未返回 W (Weight) 与 b (bias)，为了让读者更容易了解神经网络的运行情况，我们特别把 layer 函数修改为 layer_debug 函数，可返回 W 与 b ，后续可显示 W 与 b 。

1. 创建 layer_debug 函数

layer_debug 函数与 layer 类似，只是除了返回 output 之外，它还返回 W 与 b 。

```
In [40]: def layer_debug(output_dim,input_dim,inputs, activation=None):
        W = tf.Variable(tf.random_normal([input_dim, output_dim]))
        b = tf.Variable(tf.random_normal([1, output_dim]))
        XWb = tf.matmul(inputs, W) + b
        if activation is None:
            outputs = XWb
        else:
            outputs = activation(XWb)
        return outputs,W,b
```

2. 使用 layer_debug 函数建立 3 层类神经网络并显示 W 与 b

以下程序使用 layer_debug 函数建立 3 层类神经网络，并显示第一层的 $W1$ 与 $b1$ ，以及第 2 层的 $W2$ 与 $b2$ 。

```
In [15]: X = tf.placeholder("float", [None,4])
        h,W1,b1=layer_debug(output_dim=3,input_dim=4,inputs=X,
            activation=tf.nn.relu)
        y,W2,b2=layer_debug(output_dim=2,input_dim=3,inputs=h)
        with tf.Session() as sess:
            init = tf.global_variables_initializer()
            sess.run(init)
            X_array = np.array([[0.4,0.2,0.4,0.5]])
            (layer_X,layer_h,layer_y,W1,b1,W2,b2)= \
                sess.run((X,h,y,W1,b1,W2,b2),feed_dict={X:X_array})
            print('input Layer X:');print(layer_X)
            print('W1:');print( W1)
            print('b1:');print( b1)
            print('hidden Layer h:');print(layer_h)
            print('W2:');print( W2)
            print('b2:');print( b2)
            print('output Layer y:');print(layer_y)
```

以上程序代码执行结果的详细说明如图 16-4 所示。

```

input Layer X: ----- 输入层(x) 1×4
[[ 0.40000001 0.2      0.40000001 0.5      1]]
W1: ----- 权重(W1) 4×3
[[ 0.0487878 -1.69693053 0.37142262]
 [-1.01060677 -0.64186788 -0.07479379]
 [ 0.32038963 0.80540502 0.88089347]
 [-0.37085232 -0.87439799 1.48288488]]
b1: ----- 偏差(b1) 1×3
[[-0.05157181 0.39059821 1.2658087 ]]
hidden Layer h: ----- 隐藏层(h) 1×3
[[ 0.      0.      2.4932189]]
W2: ----- 权重(W2) 3×2
[[ 0.9892773 -0.23111822]
 [ 1.15233624 -1.56577241]
 [-2.8985157 -0.14242165]]
b2: ----- 偏差(b2) 1×2
[[-0.69333541 -2.36707592]]
output Layer y: ----- 输出层(y) 1×2
[[-7.91996956 -2.72216415]]

```

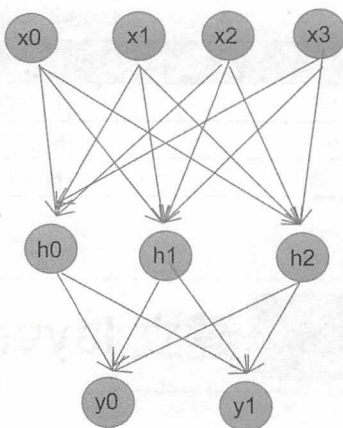


图 16-4

图 16-4 的说明如下：

- X 模拟输入层，共有 4 个神经元，我们输入的 x 值是 1×4 的张量。
- h 模拟隐藏层，共有 3 个神经元，所以是 1×3 的张量。
- y 模拟输出层，共有 2 个神经元，所以是 1×2 的张量。
- $W1$ 是权重，模拟神经元的轴突，因为输入层有 4 个神经元，隐藏层有 3 个神经元，为了让输入层与隐藏层神经元完全连接，所以 $W1$ 是 4×3 的张量。
- $b1$ 是偏差值，仿真突触的结构，代表接收神经元容易被活化的程度，偏差值越高，越容易被活化并传递信息。因为隐藏层神经元有 3 个，所以 $b1$ 是 1×3 的张量。
- $W2$ 是权重，因为隐藏层有 3 个神经元，输出层有 2 个神经元，为了让隐藏层与输出层神经元完全连接，所以 $W2$ 是 3×2 的张量。
- $b2$ 是偏差值，因为输出层神经元有 2 个，所以 $b2$ 是 1×2 的张量。

16.5 结论

在本章我们以 TensorFlow 张量（矩阵）运算来仿真类神经网络的运行，并且建立了 layer 函数，可用于构建神经网络层。接下来，第 17 章介绍用 TensorFlow 读取 MNIST 数据集，并且进行数据的预处理。

第17章

TensorFlow MNIST手写数字识别数据集

MNIST 手写数字识别数据集是由 Yann LeCun 所收集的，他也是 Convolution Neural Networks 的创始人。TensorFlow MNIST 数据集共有训练数据 55 000 项、验证数据 5 000 项、测试数据 10 000 项。每一项数据都由 features（数字图像）与 label（真实的数字）所组成，本章将介绍用 TensorFlow 读取 MNIST 数据集，并且进行数据的预处理。

本章的内容与第 6 章类似，都是下载并读取 MNIST 数据。只是第 6 章用 Keras 语句来执行，本章用 TensorFlow 语句执行，两种方式类似，但是有些地方又不相同。读者可以对照阅读，进而更加了解 TensorFlow 与 Keras。以下程序代码请参考范例程序 TensorFlow_Mnist_Introduce.ipynb。

17.1 下载 MNIST 数据

我们将建立以下 TensorFlow 程序下载并读取 MNIST 数据。

步骤01 导入 TensorFlow 模块。

```
In [1]: import tensorflow as tf
```

步骤02 导入 TensorFlow 读取 MNIST 数据集模块。

TensorFlow 已经提供了现成模块，可以用于下载并读取 MNIST 数据。

```
In [2]: import tensorflow.examples.tutorials.mnist.input_data as input_data
```

步骤03 第一次执行会下载 MNIST 数据。

第一次执行 `input_data.read_data_sets` 方法，程序会检查当前执行的目录是否有 "MNIST_data/" 目录以及是否已经有文件，如果还没有，就会下载文件。以下是第一次执行下载文件的屏幕显示界面。因为必须要下载文件，所以运行时间会比较长。

```
In [3]: mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting MNIST_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

步骤04 查看 MNIST 数据文件。

下载完成后，可以输入下列指令查看当前程序执行目录（~/pywork/TensorFlow）下的 MNIST_data 子目录。

```
ll ~/pywork/tensorFlow/MNIST_data
```

执行界面如图 17-1 所示。


```

user@Ubuntu1604: ~
user@Ubuntu1604:~$ ll ~/pywork/tensorflow/MNIST_data
总计 11336
drwxrwxrwx 1 root root 4096 1月 30 11:24 ./
drwxrwxrwx 1 root root 8192 2月 16 01:08 ../
-rwxrwxrwx 1 root root 1648877 1月 30 11:24 t10k-images-idx3-ubyte.gz*
-rwxrwxrwx 1 root root 4542 1月 30 11:24 t10k-labels-idx1-ubyte.gz*
-rwxrwxrwx 1 root root 9912422 1月 30 11:23 train-images-idx3-ubyte.gz*
-rwxrwxrwx 1 root root 28881 1月 30 11:23 train-labels-idx1-ubyte.gz*

```

图 17-1

步骤05 读取 MNIST 数据。

当我们再次执行 `input_data.read_data_sets` 时，由于之前已经下载了文件，不需要再次下载，只需要读取文件，因此运行时间不会太长。

```

In [4]: mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz

```

步骤06 查看 MNIST 数据。

下载后，可以使用下列指令来查看 MNIST 数据。

```

In [7]: print('train',mnist.train.num_examples,
            ',validation',mnist.validation.num_examples,
            ',test',mnist.test.num_examples)

train 55000 ,validation 5000 ,test 10000

```

从以上执行结果可以看到数据分为三部分。

- **train:** 训练数据 55 000 项。
- **validation:** 验证数据 5 000 项。
- **test:** 测试数据 10 000 项。

17.2 查看训练数据

先查看训练数据。

1. 训练数据是由 images 与 labels 所组成的（见图 17-2）

```

In [8]: print('train images      :', mnist.train.images.shape,
            'labels:',mnist.train.labels.shape)

train images      : (55000, 784) labels: (55000, 10)

```


➤ 定义 plot_image 函数，传入 image 作为参数

```
def plot_image(image):
```

➤ 使用 plt.imshow 显示图形

```
plt.imshow(image.reshape(28,28), cmap='binary')
```

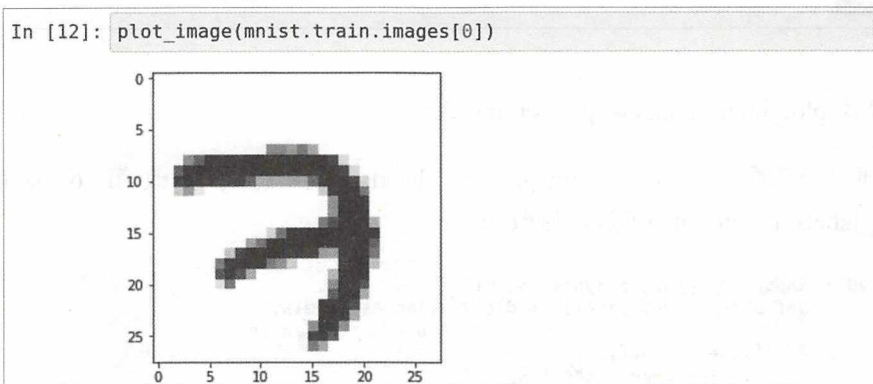
使用 plt.imshow 显示图形，传入参数 image 原本是 784 个点，必须先使用 reshape 转换为 28×28 的图形，cmap 参数设置为 binary。

➤ 开始绘图

```
plt.show()
```

5. 执行 plot_image 函数

以下程序调用 plot_image 函数传入 mnist.train.images[0]，也就是训练数据集的第 0 项数据，从显示结果中可以看到这是一个数字 7 的图形。



6. 查看训练标签 labels 数据

因为我们之前读取数据集时 mnist=input_data.read_data_sets("MNIST_data/", one_hot=True)，指定参数 one_hot 是 True，所以产生的数据 labels 是 One-Hot Encoding 格式。One-Hot Encoding 是由数字 0 与 1 所组成的，只有一个数字是 1，其余都是 0，例如：

1,0,0,0,0,0,0,0,0,0 代表 0

0,1,0,0,0,0,0,0,0,0 代表 1

0,0,1,0,0,0,0,0,0,0 代表 2

0,0,0,1,0,0,0,0,0,0 代表 3

.....

下列程序代码显示训练数据的第 0 项数据：从 0 算起第 7 个数字是 1，其余都是 0，所以此数字是 7。

```
In [13]: mnist.train.labels[0]
Out[13]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.])
```

使用 One-Hot Encoding 的原因是，后续我们要建立类神经网络的输出层，输出层共有 10 个神经元：y0~y9，分别对应 0~9。

7. 使用 argmax 显示数字

One-Hot Encoding 格式阅读起来不太方便，我们可以使用 `np.argmax` 转换为 0~9 的数字。例如下面的程序代码，将 `mnist.train.labels[0]` 转换为数字 7。

```
In [14]: import numpy as np
         np.argmax(mnist.train.labels[0])
Out[14]: 7
```

17.3 查看多项训练数据 images 与 labels

步骤01 修改 plot_images_labels_prediction() 函数。

为了便于查看多项数据 images 与 labels，我们将修改第 6 章所创建的 `plot_images_labels_prediction()` 函数，修改如下：

```
In [28]: import matplotlib.pyplot as plt
         def plot_images_labels_prediction(images, labels,
                                           prediction, idx, num=10):

             fig = plt.gcf()
             fig.set_size_inches(12, 14)
             if num>25: num=25
             for i in range(0, num):
                 ax=plt.subplot(5,5, 1+i)

                 ax.imshow(np.reshape(images[idx],(28, 28)),
                           cmap='binary')

                 title= "label=" +str(np.argmax(labels[idx]))
                 if len(prediction)>0:
                     title+=", predict="+str(prediction[idx])

                 ax.set_title(title, fontsize=10)
                 ax.set_xticks([]);ax.set_yticks([])
                 idx+=1
             plt.show()
```

主要修改了以下两部分。

(1) **转换 images 字段**：因为 TensorFlow 的 MNIST 数据集的 image 有 764 个数值，所以必须以 `np.reshape` 转换为二维 28×28 的图像才能显示出来。

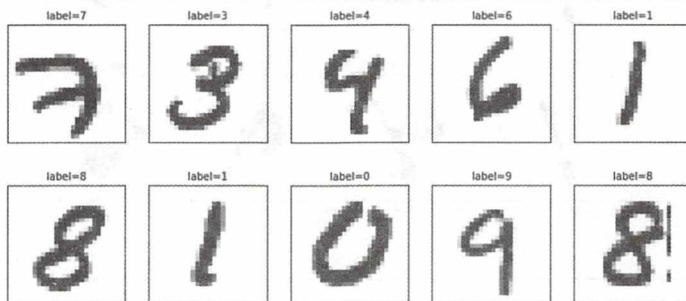
(2) **转换 labels 字段**：因为 TensorFlow 的 MNIST 数据集的 labels 字段已经是 One-Hot Encoding 格式，所以必须使用 `np.argmax` 将 One-Hot Encoding 格式转换为数字才能显示 0~9

的数字。

步骤02 查看训练数据前 10 项数据。

执行 `plot_images_labels_prediction()` 显示训练数据前 10 项数据。

```
In [38]: plot_images_labels_prediction(mnist.train.images,
                                         mnist.train.labels, [], 0)
```



步骤03 查看 validation 数据项数。

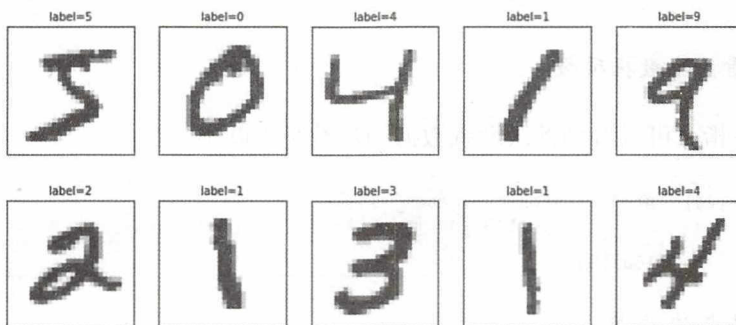
通过下面的程序代码可以看到 validation 数据项数共 5000 项。

```
In [19]: print('validation images:', mnist.validation.images.shape,
               'labels:', mnist.validation.labels.shape)
validation images: (5000, 784) labels: (5000, 10)
```

步骤04 查看 validation 数据。

执行 `plot_images_labels_prediction()` 显示验证数据前 10 项数据。

```
In [39]: plot_images_labels_prediction(mnist.validation.images,
                                         mnist.validation.labels, [], 0)
```

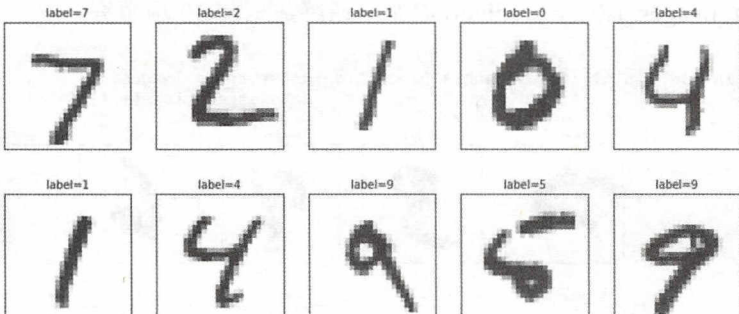


步骤05 查看 test 数据项数。

```
In [21]: print('test images:', mnist.test.images.shape,
               'labels:', mnist.test.labels.shape)
test images: (10000, 784) labels: (10000, 10)
```

步骤06 查看 test 数据。

```
In [40]: plot_images_labels_prediction(mnist.test.images,
                                         mnist.test.labels,[],0)
```



17.4 批次读取 MNIST 数据

在后续章节我们要进行深度学习网络的训练，每次训练时，并不是读取所有数据进行训练，而是读取批次数据（例如 100 项）进行训练。在 TensorFlow MNIST 模块中，已经提供了 `mnist.train.next_batch` 方法，可按批次读取数据。

步骤01 读取批次数据。

下面的程序代码使用 `mnist.train.next_batch` 方法传入参数 `batch_size=100`，每次只读取 100 项批次训练数据。读取的结果会存储在 `batch_images_xs`, `batch_labels_ys` 中。

```
In [27]: batch_images_xs, batch_labels_ys=mnist.train.next_batch(batch_size=100)
```

步骤02 查看批次数据项数。

使用以下指令可以看到批次训练数据的项数是 100 项。

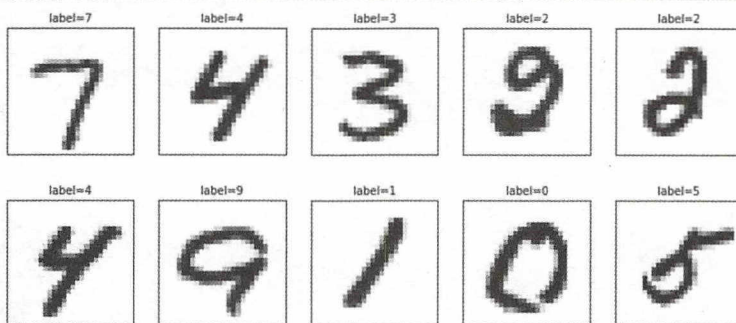
```
In [24]: print(len(batch_images_xs),
               len(batch_labels_ys))
100 100
```

步骤03 查看批次数据。

执行 `plot_images_labels_prediction()` 显示批次训练数据前 10 项数据。



```
In [41]: plot_images_labels_prediction(batch_images_xs,  
                                       batch_labels_ys,[],0)
```



17.5 结论

在本章中，我们介绍了使用 TensorFlow 下载并且读取 MNIST 数据集，还介绍了 MNIST 数据集的特色，并且完成了数据的预处理。在下一章，我们就可以使用 TensorFlow 建立多层感知器模型，并且进行训练和使用模型进行预测。

第18章

TensorFlow多层感知器 识别手写数字

本章将介绍如何使用 TensorFlow 建立多层感知器，训练模型、评估模型的准确率，然后使用训练完成的模型来识别 MNIST 手写数字，并且尝试将模型加宽、加深，以提高准确率。

本章在说明程序代码时，会比较 Keras 与 TensorFlow 在建立模型、训练模型时有哪些不同，让读者更了解这两种程序设计模式的差异。读者可以对照第 7 章 Keras 程序代码的说明。

本章完整的程序代码可参考范例程序 TensorFlow_Mnist_MLP_h256.ipynb。范例程序的下载与安装可参考本书附录 A。

18.1 TensorFlow 建立多层感知器辨识 手写数字的介绍

1. 多层感知器的训练与预测

建立如图 18-1 所示的多层感知器模型后，必须先训练才能够预测（识别）这些手写数字。

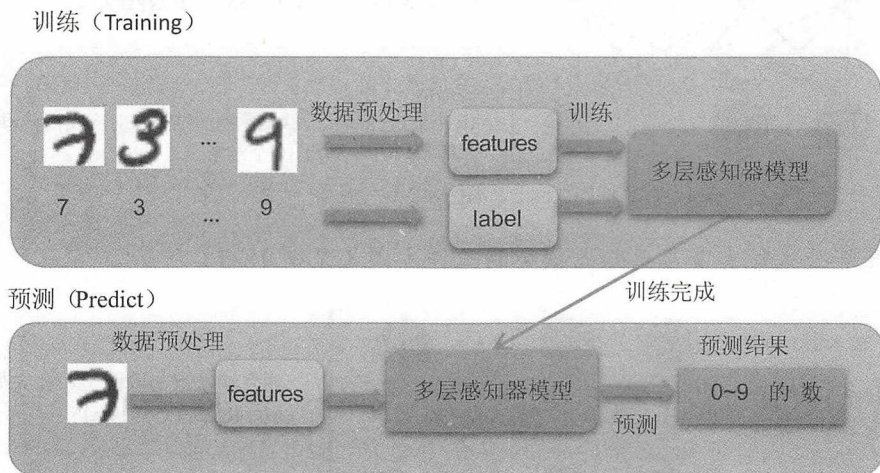


图 18-1

以多层感知器模型识别 MNIST 数字图像可分为训练与预测两个部分。

➤ 训练

MNIST 训练数据集的训练数据共 60 000 项，经过数据预处理后会产生 **features**（数字图像特征值）与 **label**（数字真实的值），然后输入多层感知器模型进行训练，训练完成的模型可以在下一阶段预测时使用。

➤ 预测

输入数字图像，预处理后会产生 **features**（数字图像转换为特征），使用训练完成的多层感知器模型进行预测，最后产生预测结果。

2. 以多层感知器模型识别 MNIST 手写数字图像

我们将以多层感知器模型识别 MNIST 手写数字图像来说明多层感知器模型的工作方式，如图 18-2 所示。

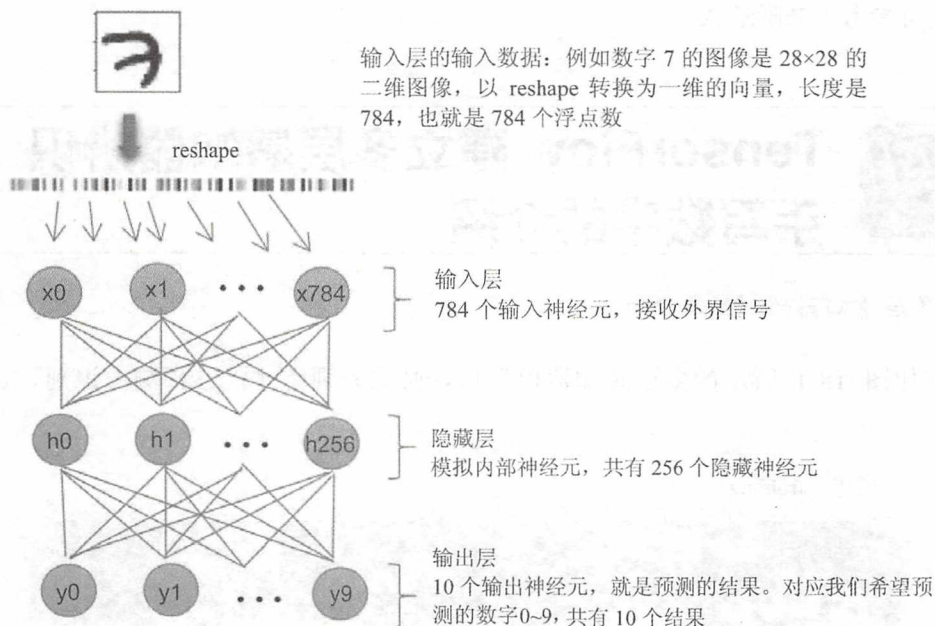


图 18-2

3. 以矩阵公式仿真多层感知器模型的工作方式（见图 18-3）

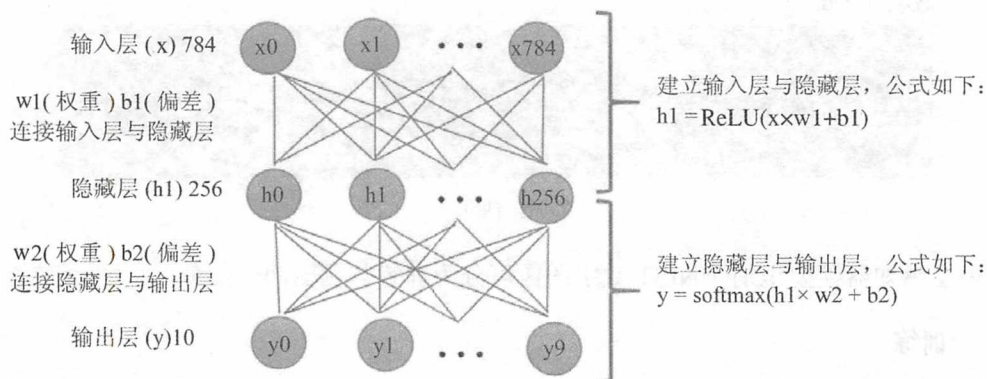


图 18-3

4. 多层感知器模型的建立步骤

多层感知器识别 MNIST 数据集中的手写数字的步骤说明如图 18-4 所示。

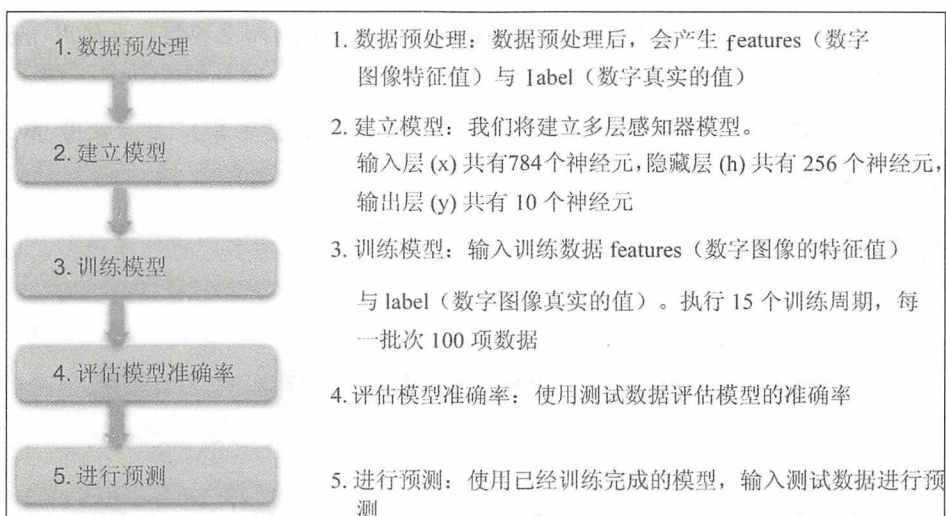


图 18-4

18.2 数据准备

首先读取 MNIST 数据集数据, 关于 MNIST 数据集的内容可参考第 17 章。

```
In [1]: import tensorflow as tf
import tensorflow.examples.tutorials.mnist.input_data as input_data

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

```
In [2]: print('train images      :', mnist.train.images.shape,
          'labels:              ', mnist.train.labels.shape)
print('validation images: ', mnist.validation.images.shape,
          'labels:              ', mnist.validation.labels.shape)
print('test images         :', mnist.test.images.shape,
          'labels:              ', mnist.test.labels.shape)

train images      : (55000, 784) labels: (55000, 10)
validation images: (5000, 784) labels: (5000, 10)
test images       : (10000, 784) labels: (10000, 10)
```

18.3 建立模型

Keras 与 TensorFlow 建立模型的方式有所不同, 说明如下。

- Keras 建立模型：只需要使用 `model=Sequential()` 建立线性堆叠模型，再使用 `model.add()` 方法将各个神经网络层加入模型即可。
- TensorFlow 建立模型：必须自行定义 `layer` 函数（处理张量运算），然后使用 `layer` 函数构建多层感知器模型。

后续的程序代码将以 TensorFlow 定义 `layer` 函数，然后构建多层感知器模型，如图 18-5 所示。

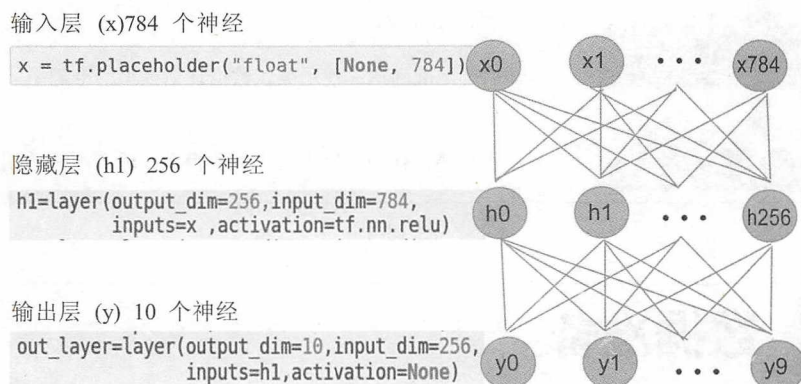


图 18-5

1. 建立 `layer` 函数

我们将使用 `layer` 函数（详细说明请参考第 16 章）构建多层感知器模型。

```
In [3]: def layer(output_dim,input_dim,inputs, activation=None):
        W = tf.Variable(tf.random_normal([input_dim, output_dim]))
        b = tf.Variable(tf.random_normal([1, output_dim]))
        XWb = tf.matmul(inputs, W) + b
        if activation is None:
            outputs = XWb
        else:
            outputs = activation(XWb)
        return outputs
```

2. 建立输入层 (x)

下列程序代码使用 `tf.placeholder` 方法建立输入层 (x)，`placeholder` 是 TensorFlow “计算图” 的输入，后续在训练时会传入数字图像数据。

```
In [5]: x = tf.placeholder("float", [None, 784])
```

建立 `tf.placeholder` 方法需设置表 18-1 中的参数。



表 18-1 建立 tf.placeholder 方法需设置的参数

参数	说明
"Float"	数据类型是 Float，即浮点数
[None, 784])	<ul style="list-style-type: none"> 第一维：设置为 None，因为后续训练时会传送很多数字图像，项数不固定，所以设置为 None 第二维：设置为 784，因为输入的数字图像是 784 像素

以上 tf.placeholder 方法会返回 x 输入层，可作为下一层的输入。

3. 建立隐藏层 (h1)

使用下列程序代码来建立隐藏层 (h1)。

```
In [7]: h1=layer(output_dim=256,input_dim=784,
               inputs=x ,activation=tf.nn.relu)
```

建立隐藏层时，调用 layer 函数需输入表 18-2 中的参数。

表 18-2 建立隐藏层时调用 layer 函数需输入的参数

参数	说明
output_dim=256	建立隐藏层神经元个数为 256
input_dim=784	x (输入层) 的神经元个数，也就是输入的数字图像像素为 784
inputs=x	x (输入层)
activation=tf.nn.relu	定义激活函数 tf.nn.relu

以上 layer 函数执行后会返回 h1 隐藏层，可作为下一层的输入。

4. 建立输出层 (y)

使用下列程序代码建立输出层 (y)。

```
In [9]: y_predict=layer(output_dim=10,input_dim=256,
                       inputs=h1,activation=None)
```

建立输出层 y_predict 时，调用 layer 函数需输入表 18-3 中的参数。

表 18-3 建立输出层 y_predict 时调用 layer 函数需输入的参数

参数	说明
output_dim=10	建立输出层神经元个数为 10
input_dim=256	h1 (隐藏层) 的神经元个数，也就是 256
inputs= h1	h1 (隐藏层)
activation= None	不需要激活函数
返回 y_predict	预测结果

18.4 定义训练方式

以下说明 Keras 与 TensorFlow 定义训练方式的不同。

- **Keras 定义训练方式：**只需要使用 `model.compile` 设置损失函数、优化器，并用 `metrics` 设置评估模型的方式。
- **TensorFlow 定义训练方式：**必须自行定义损失函数的公式、优化器和设置参数，并定义评估模型准确率的公式。

步骤01 建立训练数据 label 真实值的 placeholder。

```
In [11]: y_label = tf.placeholder("float", [None, 10])
```

以上程序代码使用 `tf.placeholder` 方法来建立 `y_label`，需设置表 18-4 中的参数。

表 18-4 使用 `tf.placeholder` 方法来建立 `y_label` 需设置的参数

参数	说明
"Float"	数据类型是 Float，即浮点数
[None, 10])	<ul style="list-style-type: none"> ● 第一维：设置为 None，因为后续我们训练时会传送很多数字图像，项数不固定，所以设置为 None ● 第二维：设置为 10，因为输入的数字真实值已经使用 One-Hot Encoding 转换，共有 10 个 0 或 1，对应数字 0~9

`placeholder` 是 TensorFlow “计算图”的输入，后续在训练时会传入数字的 label（真实值）。

步骤02 定义损失函数。

在深度学习模型的训练中使用 `cross_entropy` 交叉熵训练的效果比较好。

```
In [13]: loss_function = tf.reduce_mean(
            tf.nn.softmax_cross_entropy_with_logits
            (logits=y_predict ,
             labels=y_label))
```

以上程序代码的说明见表 18-5。

表 18-5 程序代码说明

参数	说明
<code>loss_function = tf.reduce_mean</code>	对下列 <code>cross_entropy</code> 计算结果求平均值
<code>tf.nn.softmax_cross_entropy_with_logits</code>	计算 <code>cross_entropy</code> 输入下列参数
<code>logits=y_predict</code>	<code>logits</code> 参数设置为 <code>y_predict</code> 预测值
<code>labels=y_label</code>	<code>labels</code> 参数设置为 <code>y_label</code> 真实值



步骤03 定义优化器。

```
In [15]: optimizer = tf.train.AdamOptimizer(learning_rate=0.001) \
        .minimize(loss_function)
```

以上程序代码的说明见表 18-6。

表 18-6 程序代码说明

程序代码	说明
<code>optimizer = tf.train</code>	调用 <code>tf.train</code> 模块定义 <code>optimizer</code> （优化器）
<code>.AdamOptimizer(learning_rate=0.001)</code>	使用 <code>AdamOptimizer</code> 并设置 <code>learning_rate=0.001</code>
<code>.minimize(loss_function)</code>	优化器使用 <code>loss_function</code> 计算误差，并且按照误差更新模型权重与偏差，使误差最小化

18.5 定义评估模型准确率的方式

训练模型完成后，我们希望能够评估模型的准确率。在 TensorFlow 必须定义评估模型准确率的方式。

步骤01 计算每一项数据是否预测正确。

首先，计算每一项数据是否预测正确。

```
In [17]: correct_prediction = tf.equal(tf.argmax(y_label, 1),
        tf.argmax(y_predict, 1))
```

以上程序代码的说明见表 18-7。

表 18-7 程序代码说明

程序代码	说明
<code>correct_prediction =</code>	以下运算结果存储在 <code>correct_prediction</code> 中
<code>tf.equal</code>	以 <code>tf.equal</code> 判断下列（真实值）与（预测值）是否相等，如果相等就返回 1，不相等则返回 0
<code>tf.argmax(y_label, 1)</code>	因为真实值是 One-Hot Encoding，所以使用 <code>tf.argmax</code> 转换为数字 0~9 例如，将原本的 One-Hot Encoding: 0,0,0,0,0,0,1,0,0 转换为 7
<code>tf.argmax(y_predict, 1)</code>	因为预测值是 One-Hot Encoding，所以使用 <code>tf.argmax</code> 转换为数字 0~9

步骤02 计算预测正确结果的平均值。

再将前一步的计算结果 `correct_prediction` 进行平均运算。

```
In [19]: accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

程序代码说明见表 18-8。

表 18-8 程序代码说明

程序代码	说明
<code>accuracy =</code>	以下运算结果存储在 <code>accuracy</code> 中
<code>tf.reduce_mean(tf.cast(correct_prediction, "float"))</code>	<code>correct_prediction</code> 先使用 <code>tf.cast</code> 转换为 "float", 再使用 <code>tf.reduce_mean</code> 将所有数值平均

18.6 进行训练

关于 Keras 与 TensorFlow 进行比较如下。

- **Keras 进行训练：**只需要使用 `model.fit` 就可以开始训练。
- **TensorFlow 进行训练：**必须编写程序代码来控制训练的每一个过程。

➤ 使用 TensorFlow 进行训练

以下训练数据共 55 000 项，分为每一批次 100 项，要将所有数据训练完毕需执行 550 批次（55 000/100=550 批次），当所有数据训练完毕，称为完成一个训练周期。我们将执行 15 个训练周期，尽量使误差降低，并且尽量提高准确率。整理训练过程流程图如图 18-6 所示。

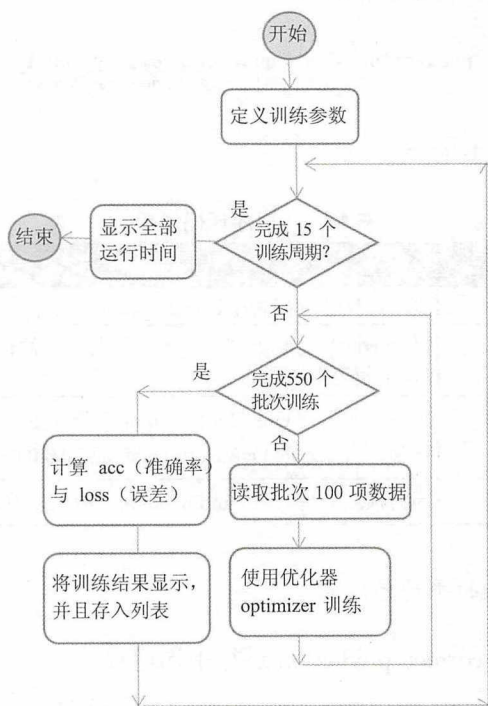


图 18-6



1. 定义训练参数

```
In [20]: trainEpochs = 15
        batchSize = 100
        loss_list=[];epoch_list=[];accuracy_list=[]
        from time import time
        startTime=time()

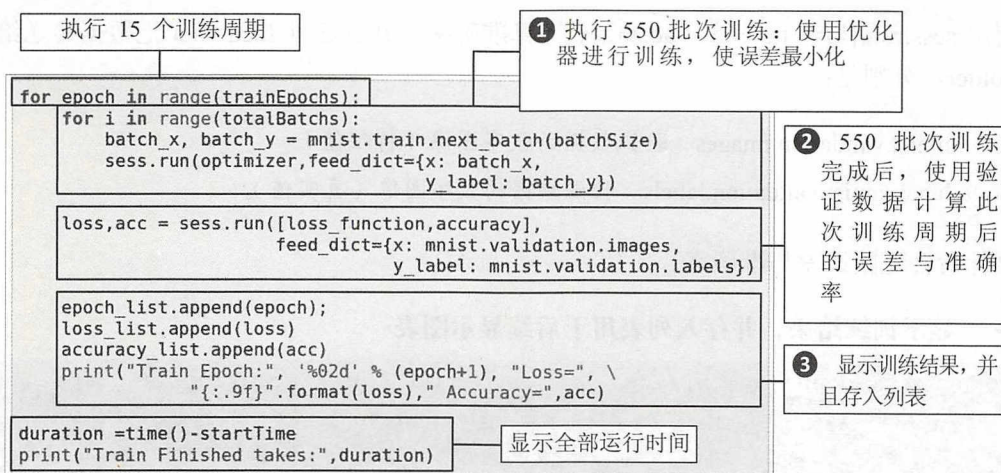
        sess = tf.Session()
        sess.run(tf.global_variables_initializer())
```

程序代码说明见表 18-9。

表 18-9 程序代码说明

程序代码	说明
trainEpochs = 15	设置执行 15 个训练周期
batchSize = 100	每一批次项数为 100
totalBatches = int(mnist.train.num_examples/ batchSize)	计算每个训练周期, (所需执行批次 550)=(训练数据项数 55000)/(每一批次项数 100)
epoch_list=[]; loss_list=[]; accuracy_list=[]	初始化训练周期 (epoch_list)、误差 (loss_list)、准确率 (accuracy_list), 后续每一个训练周期完成后, 会记录误差与准确率。在下一个步骤会以图形显示出来
from time import time startTime=time()	导入时间模块 开始计算时间
sess = tf.Session()	建立 TensorFlow Session
sess.run(tf.global_variables_initializer())	初始化 TensorFlow global 变量

2. 进行训练



以上程序代码的详细说明如下:

➤ 执行 15 个训练周期

```
for epoch in range(trainEpochs):
```

➤ 每一个训练周期执行 550 批次训练

```
for i in range(totalBatches):
    batch_x, batch_y = mnist.train.next_batch(batchSize)
    sess.run(optimizer, feed_dict={x: batch_x, y_label: batch_y})
```

for i in range(totalBatches)执行 550 批次训练。

① 读取批次数据：使用 `mnist.train.next_batch` 方法读取批次数据，传入参数 `batchSize` 是 100，每批次会读取 100 项数据，执行后返回。

- `batch_x` 数字图像（特征值）共 100 项。
- `batch_y` 数字图像（真实值）共 100 项。

② 执行批次训练：`sess.run(optimizer, feed_dict={x: batch_x, y_label: batch_y})` 使用 `sess.run` 执行优化器，通过 `feed_dict` 把数据传送给两个 `placeholder`，分别是：

- `x placeholder` 传入 `batch_x`。
- `y_label placeholder` 传入 `batch_y`。优化器按照误差值更新神经元连接的权重与偏差，尽量使损失函数的误差值最小化。

➤ 使用验证数据计算准确率

```
loss, acc = sess.run([loss_function, accuracy],
    feed_dict={x: mnist.validation.images, y_label: mnist.validation.labels})
```

使用 `sess.run([loss_function, accuracy])` 计算准确率，并且通过 `feed_dict` 把数据传送给两个 `placeholder`，分别是：

- `x: mnist.validation.images` 验证数据的数字图像（特征值）。
- `y_label: mnist.validation.labels` 验证数据的数字图像（真实值）。

执行后会返回误差与准确率。

➤ 显示训练结果，并存入列表用于后续显示图表

```
epoch_list.append(epoch); loss_list.append(loss); accuracy_list.append(acc)
print("Train Epoch:", '%02d' % (epoch+1), "Loss=", \
    "{:.9f}".format(loss), " Accuracy=", acc)
```

`epoch_list.append(epoch)` 加入训练周期列表，`loss_list.append(loss)` 加入误差列表，

`accuracy_list.append(acc)`加入准确率列表，这些列表后续可用于显示图表。

另外，使用 `print` 显示此训练周期的结果。

➤ 15 个训练周期后，计算并且显示全部训练所需的时间

```
duration = time()-startTime
print("Train Finished takes:",duration)
```

执行后结果如图 18-7 所示。

```
Train Epoch: 01 Loss= 24.219306952 Accuracy= 0.816
Train Epoch: 02 Loss= 6.276896063 Accuracy= 0.8678
Train Epoch: 03 Loss= 4.192850262 Accuracy= 0.8904
Train Epoch: 04 Loss= 3.146485436 Accuracy= 0.9058
Train Epoch: 05 Loss= 2.482951122 Accuracy= 0.9128
Train Epoch: 06 Loss= 2.009331251 Accuracy= 0.9174
Train Epoch: 07 Loss= 1.633383015 Accuracy= 0.9222
Train Epoch: 08 Loss= 1.368220688 Accuracy= 0.926
Train Epoch: 09 Loss= 1.13534218 Accuracy= 0.9288
Train Epoch: 10 Loss= 0.943261266 Accuracy= 0.9324
Train Epoch: 11 Loss= 0.796522484 Accuracy= 0.9344
Train Epoch: 12 Loss= 0.679276231 Accuracy= 0.9352
Train Epoch: 13 Loss= 0.558598945 Accuracy= 0.9394
Train Epoch: 14 Loss= 0.473082999 Accuracy= 0.9384
Train Epoch: 15 Loss= 0.397266019 Accuracy= 0.9414
Train Finished takes: 141.0081717967987
```

图 18-7

从以上执行结果的屏幕显示界面可以看到共执行了 15 个训练周期，还可以发现误差越来越小，准确率越来越高。

3. 画出误差执行结果

使用以下程序代码画出误差的执行结果。

```
In [23]: %matplotlib inline
import matplotlib.pyplot as plt
fig = plt.gcf()
fig.set_size_inches(4,2)
plt.plot(epoch_list, loss_list, label = 'loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['loss'], loc='upper left')
```

程序代码的说明见表 18-10。

表 18-10 程序代码说明

程序代码	说明
<code>%matplotlib inline</code>	设置 matplotlib 在 jupyter note 页面显示图形，如果少了这一指令，会另开窗口显示图形
<code>import matplotlib.pyplot as plt</code>	导入 matplotlib.pyplot 模块，后续会使用 plt 来引用
<code>fig = plt.gcf()</code>	获取当前的 figure 图
<code>fig.set_size_inches(4,2)</code>	设置图的大小

(续表)

程序代码	说明
<pre>plt.plot(epoch_list, loss_list, label = 'loss')</pre>	使用 plt.plot 绘图, 设置参数: x 轴数据是 epoch_list (训练周期列表) y 轴数据是 loss_list (误差列表) 线的标签是'loss'
<pre>plt.ylabel('loss')</pre>	设置 y 轴标签是'loss'
<pre>plt.xlabel('epoch')</pre>	设置 x 轴标签是'epoch'
<pre>plt.legend(['loss'],loc='upper left')</pre>	设置图例是显示'loss', 位置在左上角

显示后结果如图 18-8 所示, 我们可以看到误差越来越小。

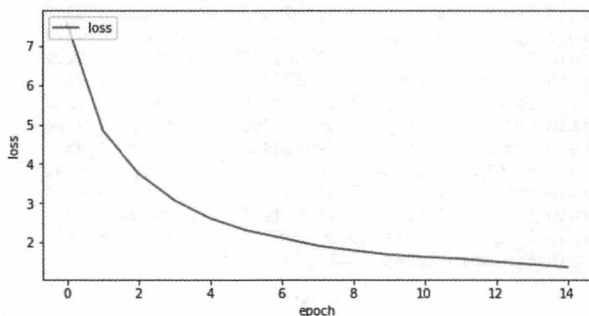


图 18-8

4. 画出准确率的执行结果

使用下面的程序代码画出准确率的执行结果。

```
In [24]: plt.plot(epoch_list, accuracy_list,label="accuracy" )
fig = plt.gcf()
fig.set_size_inches(4,2)
plt.ylim(0.8,1)
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend()
plt.show()
```

上面的程序代码与上一步类似, 只是多了 `plt.ylim(0.8,1)`, 用于设置 y 轴显示的范围, 如图 18-9 所示。

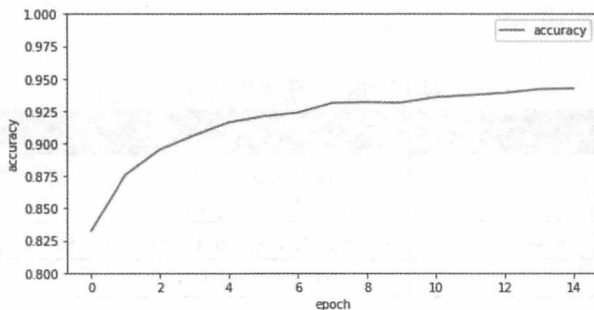


图 18-9

18.7 评估模型准确率

之前我们已经完成了训练，现在要使用 test 测试数据集，评估模型的准确率。

```
In [25]: print("Accuracy:", sess.run(accuracy,
                                         feed_dict={x: mnist.test.images,
                                                       y_label: mnist.test.labels}))
Accuracy: 0.9434
```

下面的程序代码使用 `sess.run (accuracy)` 计算准确率，并使用 `feed_dict` 把数据传送给以下两个 `placeholder`。

- `x: mnist.test.images`: 测试数据的数字图像。
- `y_label: mnist.test.labels`: 测试数据的数字真实值。

以上执行结果的准确率是 0.94。

18.8 进行预测

在之前的步骤中我们建立了模型，并且完成了模型的训练，准确率达到还可以接受的 0.94，接下来将使用此模型进行预测。

1. 执行预测

我们可以用下列指令执行预测。

```
In [26]: prediction_result=sess.run(tf.argmax(y_predict,1),
                                     feed_dict={x: mnist.test.images })
```

以上程序代码使用 `sess.run(tf.argmax(y_predict,1))` 执行预测，因为 `y_predict` 预测结果为 One-Hot Encoding 格式，所以必须使用 `tf.argmax` 转换为 0~9 的数字。执行时必须用 `feed_dict` 把数据传送给以下 `placeholder`。

- `x: mnist.test.images`: 测试数据的数字图像。

以上程序代码执行后，会将预测结果存储在 `prediction_result` 中。

2. 预测结果

可以用下列指令来查看预测结果 `prediction_result` 的前 10 项数据。

```
In [26]: prediction_result[:10]
Out[26]: array([7, 2, 1, 0, 4, 1, 4, 9, 5, 9])
```

我们可以看到第 1 项预测结果是 7，第 2 项是 2，等等。

3. 显示前 10 项预测结果

调用前一步创建的 `plot_images_labels_prediction` 函数显示前 10 项预测结果，传入测试数据图像、label 及预测结果。

```
In [29]: plot_images_labels_prediction(mnist.test.images,
mnist.test.labels,
prediction_result,0)
```

执行后的预测结果如图 18-10 所示。

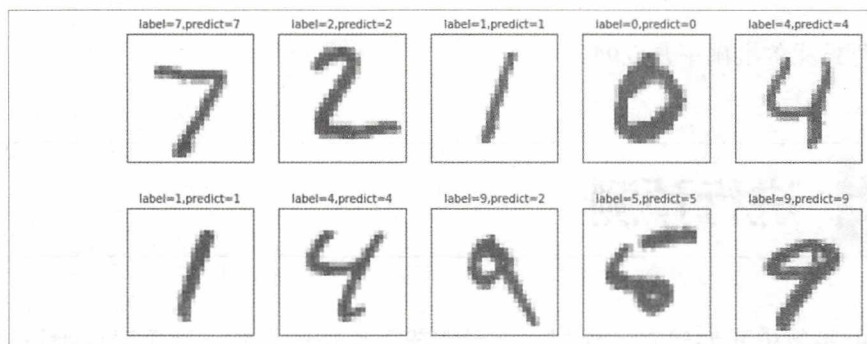


图 18-10

18.9 隐藏层加入更多神经元

为了增加多层感知器模型的准确率，在本节的范例中将隐藏层原本 256 个神经元改为 1000。本节完整的程序代码可参考范例程序 `TensorFlow_Mnist_MLP_h1000.ipynb`。

我们将使用下面的程序代码建立多层感知器模型。

步骤01 修改隐藏层原本 256 个神经元为 1000 个神经元。


```

In [4]: # 建立输入层 x
In [5]: x = tf.placeholder("float", [None, 784])
In [6]: # 建立隐藏层h1
In [7]: h1=layer(output_dim=1000,input_dim=784,
                  inputs=x,activation=tf.nn.relu)
In [8]: # 建立输出层
In [9]: y_predict=layer(output_dim=10,input_dim=1000,
                       inputs=h1,activation=None)
In [10]: # 建立训练数据 label 真实值 placeholder
In [11]: y_label = tf.placeholder("float", [None, 10])

```

原本 256 改为 1000

步骤02 预测准确率。

```

In [22]: print("Accuracy:", sess.run(accuracy,
                                     feed_dict={x: mnist.test.images,
                                               y_label: mnist.test.labels}))

Accuracy: 0.9546

```

以上执行结果的准确率是 0.95，比上一节模型的准确率稍微提高了。

18.10 建立包含两个隐藏层的多层感知器模型

为了更进一步增加多层感知器模型的准确率，在本节范例中将建立两个隐藏层。本节完整的程序代码可参考范例程序 TensorFlow_Mnist_MLP_h1000-h1000.ipynb。

如图 18-11 所示，我们将使用下面的程序代码加入两个隐藏层。

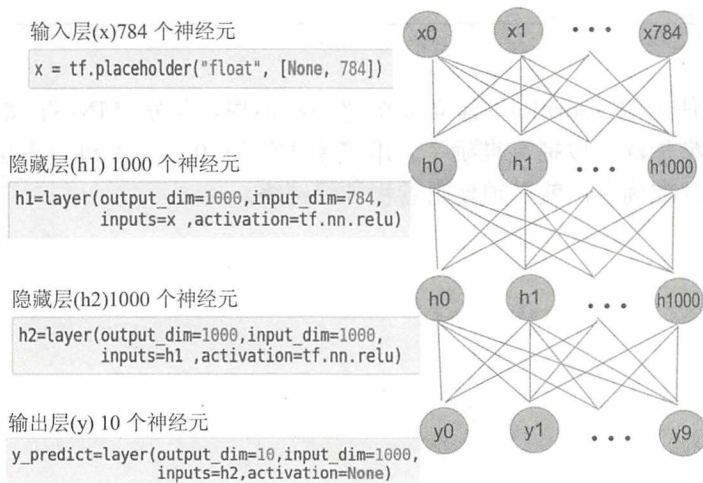


图 18-11

步骤01 建立多层感知器模型，包含两个隐藏层。

```
In [4]: # 建立输入层 x
In [5]: x = tf.placeholder("float", [None, 784])
In [6]: # 建立隐藏层h1
In [7]: h1=layer(output_dim=1000,input_dim=784,
                 inputs=x ,activation=tf.nn.relu)
In [8]: # 建立隐藏层h2
In [9]: h2=layer(output_dim=1000,input_dim=1000,
                 inputs=h1 ,activation=tf.nn.relu)
In [10]: # 建立输出层
In [11]: y_predict=layer(output_dim=10,input_dim=1000,
                        inputs=h2,activation=None)
In [12]: # 建立训练数据 label 真实值 placeholder
```

增加隐藏层 2

步骤02 预测准确率。

```
In [26]: print("Accuracy:", sess.run(accuracy,
                                     feed_dict={x: mnist.test.images,
                                               y_label: mnist.test.labels}))
Accuracy: 0.9658
```

以上执行结果的准确率是 0.96，比上一节模型的准确率又稍微提高了。

18.11 结论

在本章中我们使用 TensorFlow 建立多层感知器模型，识别 MNIST 数据集中的手写数字，并且尝试将模型加深，以提高准确率。准确率大约为 0.96。不过，多层感知器有极限，如果还要进一步提升准确率，就必须使用卷积神经网络。

第19章

TensorFlow卷积神经网络 识别手写数字

在前面的章节中，我们使用多元感知器识别 MNIST 数据集中的手写数字，准确率大约是 0.96。在本章节中，我们将使用卷积神经网络（Convolutional Neural Network, CNN）来识别 MNIST 数据集中的手写数字，其分类精度接近 0.99。

卷积神经网络是由一位计算机科学家 Yann LeCun 所提出的。他在机器学习、计算机视觉和计算神经科学等诸多领域都有不少贡献。

本章完整的程序代码可参考范例程序 TensorFlow_Mnist_CNN.ipynb。有关范例程序的下载与安装可参考本书附录 A 中的“本书范例程序的下载与安装说明”。

19.1 卷积神经网络简介

1. 卷积神经网络的介绍

卷积层的意义是，将原本一个图像经过卷积运算产生多个图像，就好像卷积起来。卷积神经网络可分为两大部分（见图 19-1）：

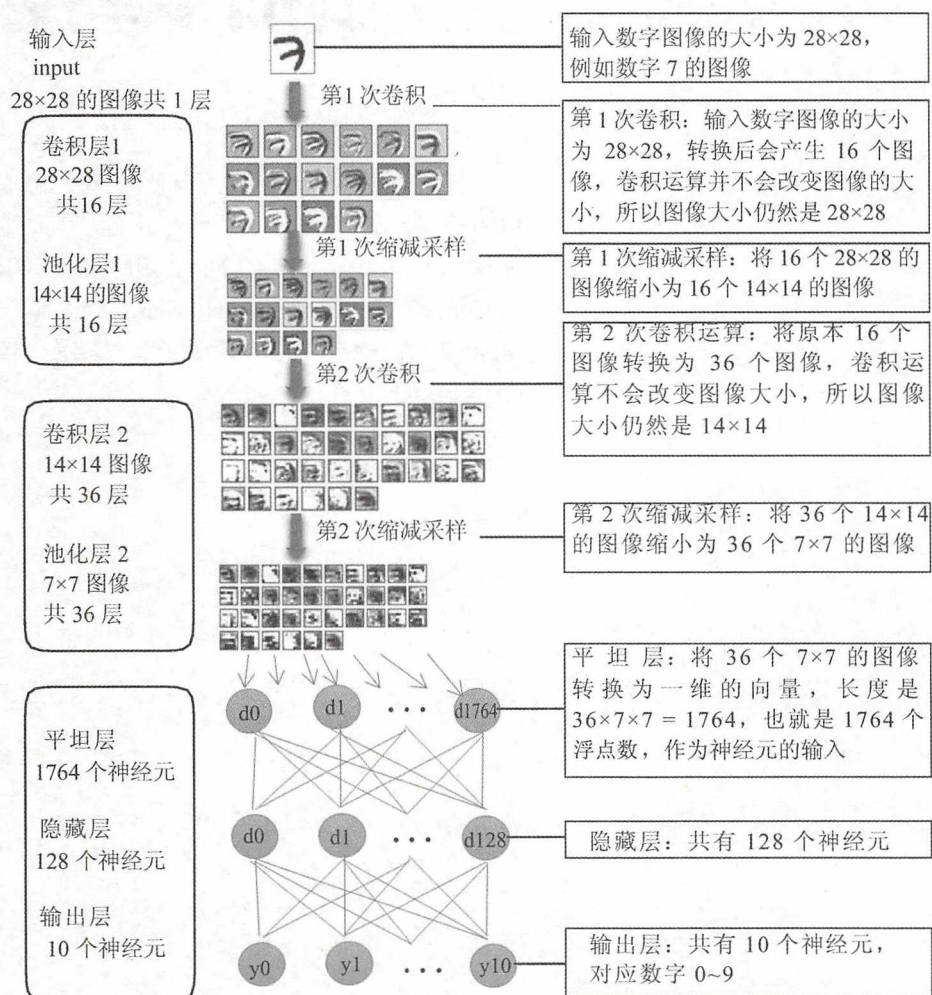


图 19-1

● “卷积”与“缩减采样”提取图像的特征

通过“第 1 次卷积”与“第 1 次缩减采样”、“第 2 次卷积”与“第 2 次缩减采样”的处理来提取图像的特征。通过以上这种方式可以提高识别的准确率。

● 完全连接神经网络

提取图像的特征后，reshape 转换为一维的向量，送入由“平坦层”“隐藏层”“输出层”所组成的类神经网络进行处理。

本章在说明程序代码时，会比较 Keras 与 TensorFlow 建立模型、训练模型的不同，让读者更了解这两种程序设计模式的差异，可以对照第 8 章有关 Keras 程序代码的说明。

2. 建立卷积神经网络识别 MNIST 数据集的步骤

建立卷积神经网络识别 MNIST 数据集的步骤如图 19-2 所示。

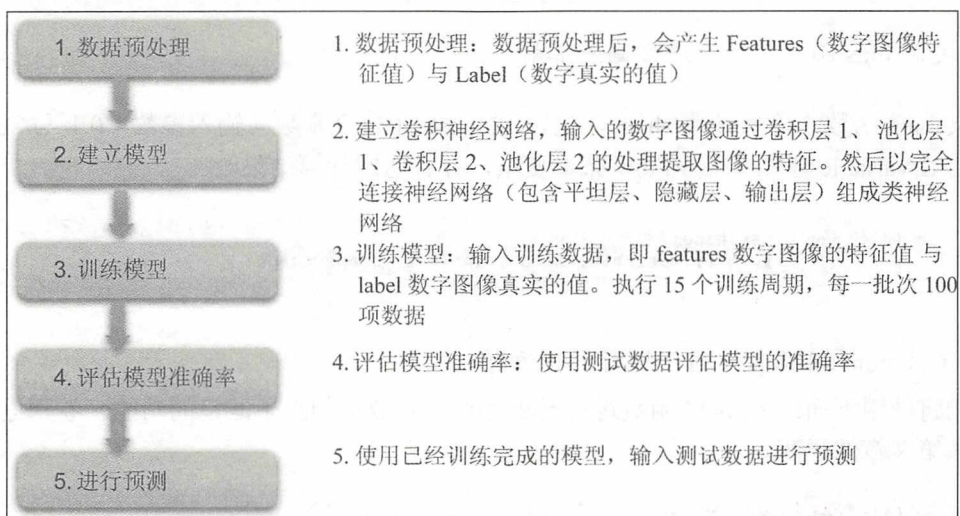


图 19-2

19.2 进行数据预处理

使用下面的程序代码读取 MNIST 数据集，可参考第 17 章的说明。

```
In [1]: import tensorflow as tf
import tensorflow.examples.tutorials.mnist.input_data as input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

19.3 建立共享函数

为了便于后续建立模型，我们建立下面的共享函数。

1. 定义 weight 函数，用于建立权重张量

定义 weight 函数，输入参数 shape，先以 tf.truncated_normal 随机方式初始化权重，然后使用 tf.Variable 建立 TensorFlow 变量。

```
In [2]: def weight(shape):  
        return tf.Variable(tf.truncated_normal(shape, stddev=0.1),  
                           name='W')
```

2. 定义 bias 函数，用于建立偏差张量

定义 bias 函数，输入参数 shape，先以 tf.constant 建立常数（输入参数：0.1 与 shape），然后使用 tf.Variable 建立 TensorFlow 张量变量，并且返回计算结果。

```
In [3]: def bias(shape):  
        return tf.Variable(tf.constant(0.1, shape=shape),  
                           name='b')
```

3. 定义 conv2d 函数，用于进行卷积运算

我们将使用下面的 conv2d 函数进行卷积运算，其效果相当于滤镜的功能。卷积运算的细节可参考第 8 章的说明。

```
In [4]: def conv2d(x, W):  
        return tf.nn.conv2d(x, W, strides=[1,1,1,1],  
                             padding='SAME')
```

使用 TensorFlow 提供的 tf.nn.conv2d 函数进行卷积运算，并且返回运算结果，执行时需输入下列参数。

- **x 是输入的图像**：后续我们会传入要处理的图像，必须是四维的张量。
- **W 是 filter weight 滤镜的权重**：后续我们会以随机方式产生 filter weight 并且传给此参数。
- **strides**：滤镜的步长，设置为[1, 1, 1, 1]，其格式是[1, stride, stride, 1]，也就是滤镜每次移动时，从左到右、从上到下各一步。
- **padding**：设置为'SAME'模式，此模式会在边界之外补 0，在进行运算时，让输入与输出图像的大小相同。

卷积运算的结果如图 19-3 所示。

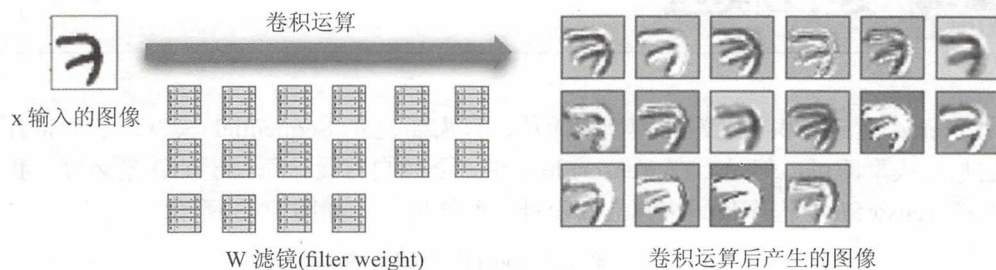


图 19-3

4. 出具 max_pool_2x2 函数，用于建立池化层

我们将创建 max_pool_2x2 函数，用于建立池化层进行图像的缩减采样。关于 Max-Pool 运算的细节，请参考第 8 章的说明。

```
In [5]: def max_pool_2x2(x):
        return tf.nn.max_pool(x, ksize=[1,2,2,1],
                               strides=[1,2,2,1],
                               padding='SAME')
```

使用 TensorFlow 提供的 tf.nn.max_pool 函数建立池化层并且返回计算结果，执行时需输入下列参数。

- **x 是输入的图片**：后续我们会传入要处理的图像，必须是四维的张量。
- **ksize**：缩减采样窗口的大小，设置为[1, 2, 2, 1]，其格式是[1, height, width, 1]，也就是高度=2、宽度=2的窗口。
- **strides**：缩减采样窗口的跨步，设置为[1, 2, 2, 1]，其格式是[1, stride, stride, 1]，也就是缩减采样窗口，从左到右、从上到下移动时的步长各两步。

后续处理手写数字图像时，原本 28×28 的图像经过 Max-Pool 之后，缩小为 14×14 的图像，如图 19-4 所示。

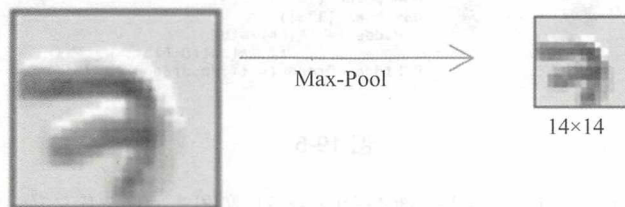


图 19-4

19.4 建立模型

之前 Keras 建立卷积神经网络模型很简单，只需要建立 Sequential 模型，然后将各个神经网络层加入模型即可，但是，在 TensorFlow 中，必须自行设计每一层的张量运算。我们将使用下面的 TensorFlow 程序代码来建立卷积神经网络模型，如图 19-5 所示。

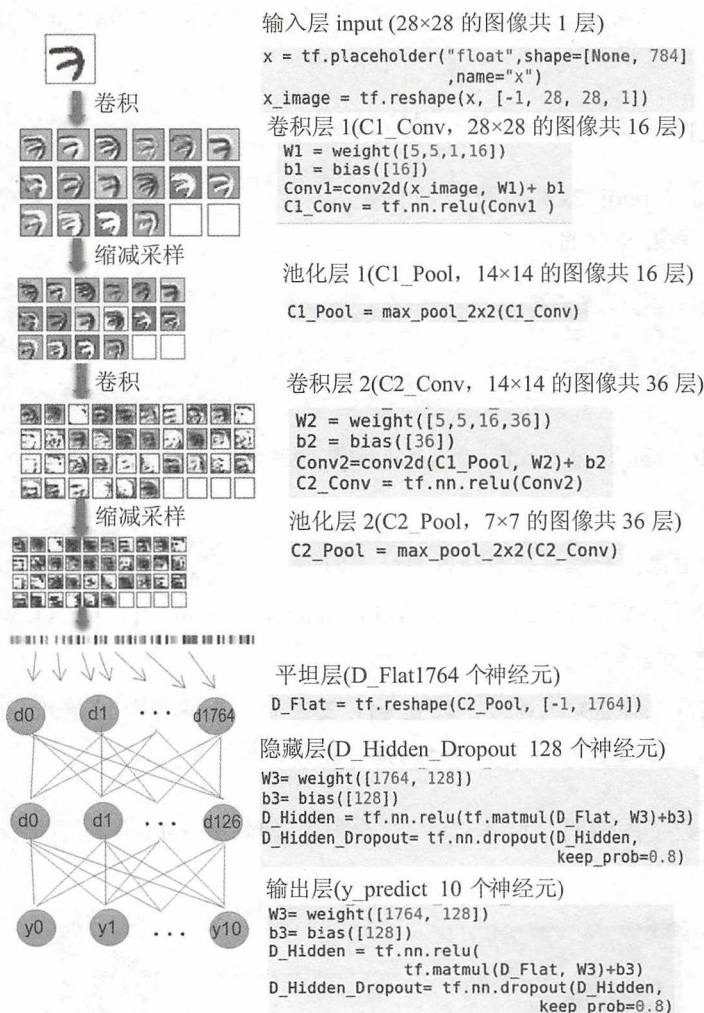


图 19-5

在下面的程序代码中，每一个层级都会加入“计算图”的层级名称。例如，输入层会加入 with tf.name_scope('Input_Layer') 的程序代码，其功能是设置输入层的名称为 'Input_Layer'。

如果不设置“计算图”的层级名称，程序仍然可以正常运行，但是因为 CNN 的层级比较复杂而且层级很多，所以设置层级名称可以让程序代码比较易读。另外，后续当我们使用 TensorBoard 查看“计算图”时，可以很清楚地看到每一个层级，比较容易使用 TensorBoard



查看“计算图”，会在后面的章节介绍。

1. 输入层

使用下面的程序代码建立输入层。

```
In [6]: with tf.name_scope('Input_Layer'):  
        x = tf.placeholder("float", shape=[None, 784]  
                           , name="x")  
        x_image = tf.reshape(x, [-1, 28, 28, 1])
```

以上程序代码使用 `with tf.name_scope('Input_Layer')` 设置“计算图”输入层的名称，其余程序代码说明如下：

➤ 建立输入层 (x)

```
x = tf.placeholder("Float", shape=[None, 784], name="x")
```

`placeholder` 是 TensorFlow “计算图”的输入，后续在训练时会传入数字图像数据。

建立 `tf.placeholder` 方法需设置下列参数。

- **"Float"**：数据类型是 `Float`。
- **shape=[None, 784]**：第一维设置为 `None`，因为后续我们训练时会传送很多数字图像，项数不固定，所以设置为 `None`。第二维设置为 `784`，因为输入的数字图像像素是 `784`。

➤ x reshape 为四维张量

```
x_image = tf.reshape(x, [-1, 28, 28, 1])
```

`x` 原本是一维向量，因为后续要进行卷积与池化运算，所以必须转换为四维张量，说明如下。

- **第一维是-1**：因为后续训练时通过 `placeholder` 输入的项数不固定，所以设置为 `-1`。
- **第二、三维是 28, 28**：输入的数字图像大小是 `28×28`。
- **第四维是 1**：因为是单色，所以设置为 `1`，如果是彩色，就要设置为 `3`。

2. 建立卷积层 1

卷积层的运算会以单个图像来产生多个图像，卷积运算后的效果类似于滤镜效果。这有助于提取输入的不同特征，例如边缘、线条和角等。

输入数字图像的大小为 `28×28`，例如数字 `7` 的图像。卷积运算后会产生 `16` 个图像，卷积运算并不会改变图像大小，所以图像大小仍然是 `28×28`。我们可以看到这些图像仍然像 `7`，但是提取了不同的特征。

使用下面的程序代码建立卷积层 1，计算图的层级名称是 `C1_Conv`。



```
In [7]: with tf.name_scope('C1_Conv'):  
        W1 = weight([5,5,1,16])  
        b1 = bias([16])  
        Conv1=conv2d(x_image, W1)+ b1  
        C1_Conv = tf.nn.relu(Conv1 )
```

以上程序代码的详细说明如下：

➤ 建立 $W1$ 权重

```
W1 = weight([5,5,1,16])
```

使用之前建立的 `weight` 共享函数建立 $W1$ 权重，共有四维，说明如下。

- **第一、二维均是 5：**代表滤镜（filter weight）的大小为 5×5 。
- **第三维是 1：**因为数字图像是单色的，所以设置为 1，如果是彩色的，就要设置为 3。
- **第四维是 16：**要产生 16 个图像。

➤ 建立 $b1$ 偏差值

```
b1 = bias([16])
```

使用之前建立的 `bias` 函数建立偏差值 $b1$ 。因为卷积层 1 要产生 16 个图像，所以输入参数 `shape=[16]`。

➤ 进行卷积运算

```
Conv1 = conv2d(x_image, W1) + b1
```

使用之前建立的 `conv2d` 函数进行卷积计算，输入参数：`x_image`（要处理的图像）、 $W1$ （滤镜的权重）、`conv2d` 计算结果以及偏差值。

➤ ReLU 激活函数

```
C1_Conv = tf.nn.relu(Conv1 )
```

以上卷积运算的结果再由 ReLU 激活函数转换，最后的结果是 `C1_Conv`。

3. 建立池化层 1

池化层使用缩减采样会将图像由 28×28 缩小为 14×14 ，不会改变图像数量（仍然是 16）。

缩减采样会缩小图像，有下列好处。

- (1) **减少所需处理的数据点：**减少后续运算所需的时间。
- (2) **让图像位置差异变小：**例如手写数字 7，位置上下左右可能不同，但是位置的不同可能会影响识别，减小图像的大小让数字的位置差异变小。
- (3) **参数的数量和计算量下降：**这在一定程度上也控制了过度拟合。



下面的程序代码使用之前创建的 `max_pool_2x2` 函数传入卷积层 `C1_Conv` 进行缩减采样，建立池化层 1，计算图的层级名称是 `C1_Pool`。

```
In [8]: with tf.name_scope('C1_Pool'):
        C1_Pool = max_pool_2x2(C1_Conv)
```

4. 建立卷积层 2

第 2 次卷积运算将原本的 16 个图像转换为 36 个图像，卷积运算不会改变图像的大小，所以图像的大小仍然是 14×14 。使用下面的程序代码建立卷积层 2，计算图的层级名称是 `C2_Conv`。

```
In [9]: with tf.name_scope('C2_Conv'):
        W2 = weight([5,5,16,36])
        b2 = bias([36])
        Conv2=conv2d(C1_Pool, W2)+ b2
        C2_Conv = tf.nn.relu(Conv2)
```

以上程序代码的详细说明如下：

➤ 建立 $W2$ 权重

```
W2 = weight([5,5,16,36])
```

使用之前创建的 `weight` 共享函数建立 $W2$ 权重，共有四维，说明如下。

- 第一、二维均是 5：代表滤镜的大小为 5×5 。
- 第三维是 16：因为卷积层 1 的图像数量是 16 个。
- 第四维是 36：因为要将原本的 16 个图像转换为 36 个图像。

➤ 建立偏差值向量

```
b2 = bias([36])
```

使用之前创建的 `bias` 函数建立偏差值 $b2$ 。因为卷积层 2 要产生 36 个图像，所以输入参数 `shape=[36]`。

➤ 进行卷积运算

```
Conv2=conv2d(C1_Pool, W2)+ b2
```

使用之前创建的 `conv2d` 共享函数进行卷积计算，传入参数：`C1_Pool`（池化层 1）、 $W2$ （filter weight）以及偏差值向量 $b2$ 。

➤ ReLU 激活函数

```
C2_Conv = tf.nn.relu(Conv2)
```

最后由 ReLU 激活函数转换，ReLU 会将原本是负数的点转换为 0。

5. 建立池化层 2

下面的程序代码使用之前创建的 `max_pool_2x2` 函数传入 `C2_Conv`（卷积层 2）进行缩减采样，建立池化层 2，计算图的层级名称是 `C2_Pool`。

```
In [10]: with tf.name_scope('C2_Pool'):
         C2_Pool = max_pool_2x2(C2_Conv)
```

6. 建立平坦层

平坦层可以将池化层 2 的 36 个 7×7 的图像转换为一维的向量，长度是 $36 \times 7 \times 7 = 1764$ ，也就是 1764 个浮点数，作为神经元的输入。使用下面的程序代码建立平坦层，计算图的层级名称是 `D_Flat`。

```
In [11]: with tf.name_scope('D_Flat'):
         D_Flat = tf.reshape(C2_Pool, [-1, 1764])
```

以上程序代码使用 `tf.reshape` 传入下列参数。

- **C2_Pool**: 此参数设置为要进行 reshape 的张量。
- **[-1, 1764]** :
 - 第一维是 -1，因为后续会传入不限项数的训练数据——数字图像。
 - 第二维是 1764，因为 `C2_Pool` 是 36 个 7×7 的图像，要转换为一维的向量，长度是 $36 \times 7 \times 7 = 1764$ 。

7. 建立隐藏层

使用下面的程序代码建立隐藏层，计算图的层级名称是 `D_Hidden_Layer`。

```
In [12]: with tf.name_scope('D_Hidden_Layer'):
         W3= weight([1764, 128])
         b3= bias([128])
         D_Hidden = tf.nn.relu(
             tf.matmul(D_Flat, W3)+b3)
         D_Hidden_Dropout= tf.nn.dropout(D_Hidden,
             keep_prob=0.8)
```

以上程序代码详细说明如下：

➤ 建立 W3 权重

```
W3= weight([1764, 128])
```

使用 `weight` 共享函数建立 `W3` 权重，输入 `shape` 参数，说明如下。

- 第一维是 1764，因为上一层 `D_Flat` 有 1764 个神经元。



- **第二维是 128**，因为要建立的隐藏层 D_Hidden 有 128 个神经元。

➤ 建立偏差值向量

```
b3= bias([128])
```

使用之前创建的 bias 函数建立偏差值 b3。因为要建立的隐藏层 D_Hidden 有 128 个神经元，所以输入参数 shape=[128]。

➤ 建立隐藏层 (D_Hidden_Layer)

```
D_Hidden = tf.nn.relu(tf.matmul(D_Flat, W3)+b3)
```

建立隐藏层的公式如下：

```
D_Hidden = relu(D_Flat×W3 + b3)
```

先使用 tf.matmul 将 D_Flat 与 W3 矩阵相乘，再加上偏差值向量，最后使用 tf.nn.relu 激活函数转换后，就可以得到隐藏层 D_Hidden。

➤ 加入 Dropout 避免过度拟合

```
D_Hidden_Dropout= tf.nn.dropout(D_Hidden, keep_prob=0.8)
```

tf.nn.dropout 的功能是，每次训练迭代时都会随机地在神经网络中放弃一些神经元，以避免过度拟合。输入的参数如下。

- **D_Hidden**：要执行 dropout 的神经网络层。
- **keep_prob=0.8**：设置要保留的神经元比率，0.8 代表要保留 80% 的神经元，随机去掉 20% 的神经元。

8. 建立输出层

输出层共有 10 个神经元，对应数字 0~9。建立隐藏层的公式如下：

```
y_predict = softmax(D_Hidden_Dropout×W4 + b4)
```

下面的程序代码建立隐藏层，计算图的层级名称是 Output_Layer。

```
In [13]: with tf.name_scope('Output_Layer'):
          W4 = weight([128,10])
          b4 = bias([10])
          y_predict= tf.nn.softmax(
                        tf.matmul(D_Hidden_Dropout,
                                  W4)+b4)
```

以上程序代码的详细说明如下：

➤ 建立 W4 权重

```
W4= weight([128,10])
```

使用 weight 共享函数建立 W4 权重，输入 shape 参数，说明如下。

- 第一维是 128，因为上一层 D_Hidden 有 128 个神经元。
- 第二维是 10，因为要建立的输出层（Output_Layer）有 10 个神经元。

➤ 建立偏差值向量

```
b4= bias([10])
```

使用之前创建的 bias 函数建立偏差值 b4。因为要建立的输出层（Output_Layer）有 10 个神经元，所以输入参数 shape=[10]。

➤ 建立输出层（y_predict）

```
y_predict= tf.nn.softmax(tf.matmul(D_Hidden_Dropout,W4)+b4)
```

先使用 tf.matmul 将 D_Hidden_Dropout 与 W4 矩阵相乘，再加上偏差值向量 b4，最后使用 tf.nn.softmax 激活函数转换后，就可以得到输出层 y_predict。

19.5 定义训练方式

在之前的步骤中已经建立了卷积神经网络模型。接下来，使用反向传播算法训练多层感知器模型。

下面的程序代码与第 18 章的程序代码完全相同，详细说明可参考第 18 章。

```
In [14]: with tf.name_scope("optimizer"):
          y_label = tf.placeholder("float", shape=[None, 10],
                                   name="y_label")

          loss_function = tf.reduce_mean(
              tf.nn.softmax_cross_entropy_with_logits
              (logits=y_predict,
               labels=y_label))

          optimizer = tf.train.AdamOptimizer(learning_rate=0.0001) \
              .minimize(loss_function)
```

19.6 定义评估模型准确率的方式

当使用上一节的方法训练模型完成某一阶段后，我们希望能够评估模型的准确率。

下面的程序代码与第 18 章的程序代码完全相同，详细说明可参考第 18 章。

```
In [15]: with tf.name_scope("evaluate model"):
          correct_prediction = tf.equal(tf.argmax(y_predict, 1),
                                       tf.argmax(y_label, 1))
          accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

19.7 进行训练

我们将使用反向传播算法训练，训练数据共 55 000 项，分为每一批次 100 项，要将所有数据训练完毕需执行 550 批次（55 000/100=550 批次），当所有数据训练完毕后，完成一个训练周期。

我们将执行 30 个训练周期，尽量使误差降低，并且尽可能提高准确率。

下面的程序代码与第 18 章的程序代码完全相同，详细说明可参考第 18 章。

步骤01 定义训练参数。

```
In [16]: trainEpochs = 30
          batchSize = 100
          totalBatches = int(mnist.train.num_examples/batchSize)
          epoch_list=[];accuracy_list=[];loss_list=[];
          from time import time
          startTime=time()
          sess = tf.Session()
          sess.run(tf.global_variables_initializer())
```

步骤02 进行训练。



结果如图 19-6 所示。


```

Train Epoch: 20 Loss= 1.477607131 Accuracy= 0.986
Train Epoch: 21 Loss= 1.479106903 Accuracy= 0.9834
Train Epoch: 22 Loss= 1.477838755 Accuracy= 0.9836
Train Epoch: 23 Loss= 1.478707433 Accuracy= 0.9848
Train Epoch: 24 Loss= 1.476776361 Accuracy= 0.9856
Train Epoch: 25 Loss= 1.477888465 Accuracy= 0.9848
Train Epoch: 26 Loss= 1.476577163 Accuracy= 0.9854
Train Epoch: 27 Loss= 1.476924658 Accuracy= 0.9856
Train Epoch: 28 Loss= 1.477100253 Accuracy= 0.9844
Train Epoch: 29 Loss= 1.477174640 Accuracy= 0.9852
Train Epoch: 30 Loss= 1.476072311 Accuracy= 0.9864
Train Finished takes: 3640.4850730895996

```

图 19-6

从训练结果可知准确率达到 0.9864。

步骤03 画出误差执行的结果，如图 19-7 所示。

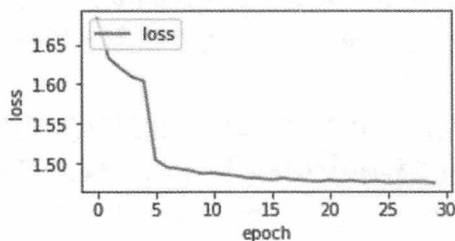


图 19-7

步骤04 画出准确率执行的结果，如图 19-8 所示。

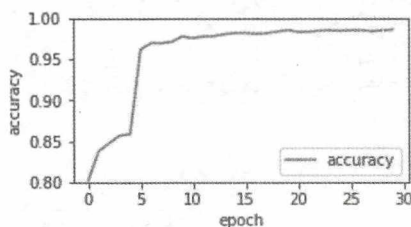


图 19-8

19.8 评估模型准确率

之前我们已经完成训练，现在要使用 `test` 测试数据集评估模型的准确率。评估模型的准确率与第 18 章的程序代码完全相同，可参考第 18 章的说明。以下仅列出执行的结果。

```

In [21]: print("Accuracy:",
            sess.run(accuracy, feed_dict={x: mnist.test.images,
                                           y_label: mnist.test.labels}))

Accuracy: 0.9866

```

19.9 进行预测

在之前的步骤中我们建立了模型，并且完成了模型的训练，准确率达到还可以接受的 0.98，接下来将使用此模型进行预测。下面的程序代码与第 18 章的程序代码完全相同，详细说明可参考第 18 章。

1. 执行预测

我们可以用下面的指令执行预测。

```
In [26]: prediction_result=sess.run(tf.argmax(y_predict,1),
                                     feed_dict={x: mnist.test.images ,
                                                  y_label: mnist.test.labels})
```

这段程序代码使用了 `sess.run(tf.argmax(y_predict,1))`，因为 `y_predict` 是 One-Hot Encoding，所以必须先使用 `tf.argmax` 进行转换，再进行预测并使用 `feed_dict` 传入。

- `x: mnist.test.images`: 测试数据的数字图像。

2. 预测结果

我们可以用下列指令来查看预测结果的前 10 项数据。

```
In [27]: prediction_result[:10]
Out[27]: array([7, 2, 1, 0, 4, 1, 4, 9, 5, 9])
```

可以看到第 1 项预测结果是 7，第 2 项是 2……

3. 显示前 10 项预测结果

在第 18 章中创建的 `show_images_labels_prediction` 函数中显示前 10 项预测结果，传入测试数据图像、label 及预测结果。

```
In [29]: plot_images_labels_prediction(mnist.test.images,
                                       mnist.test.labels,
                                       prediction_result,0)
```

执行后预测结果如图 19-9 所示。

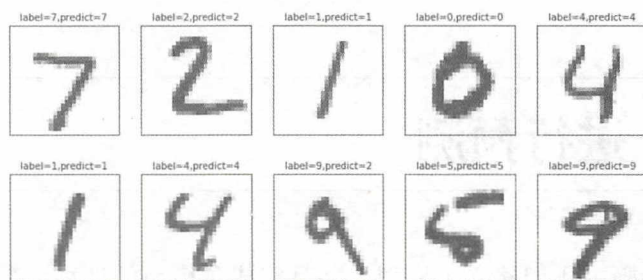


图 19-9

19.10 TensorBoard

TensorFlow 提供了 TensorBoard，可以让我们以可视化的方式来查看所建立的“计算流程图”，我们可按照下列步骤来查看卷积神经网络的“计算图”。

首先，使用程序代码将要显示在 TensorBoard 的计算图写入 log 文件。

```
In [36]: merged = tf.summary.merge_all()
         train_writer = tf.summary.FileWriter('log/CNN', sess.graph)
```

1. 在 Windows 中启动 TensorBoard

如果读者使用的是 Windows 系统，就按照下列步骤启动 TensorBoard。启动“命令提示符”程序，并且输入下列命令。

➤ 先确认 log 目录文件是否已经产生

在 Windows 的“命令提示符”程序中使用 dir 显示目录。

```
dir c:\pythonwork\tensorFlow\log\CNN
```

➤ 启用 TensorFlow 的 Anaconda 虚拟环境

```
activate tensorflow
```

➤ 启动 TensorBoard

启动 TensorBoard 指令需指定 log 文件目录，TensorBoard 会读取此目录，并显示在 TensorBoard 上。

```
tensorboard --logdir=c:\pythonwork\tensorFlow\log\CNN
```

执行后屏幕显示界面如图 19-10 所示。

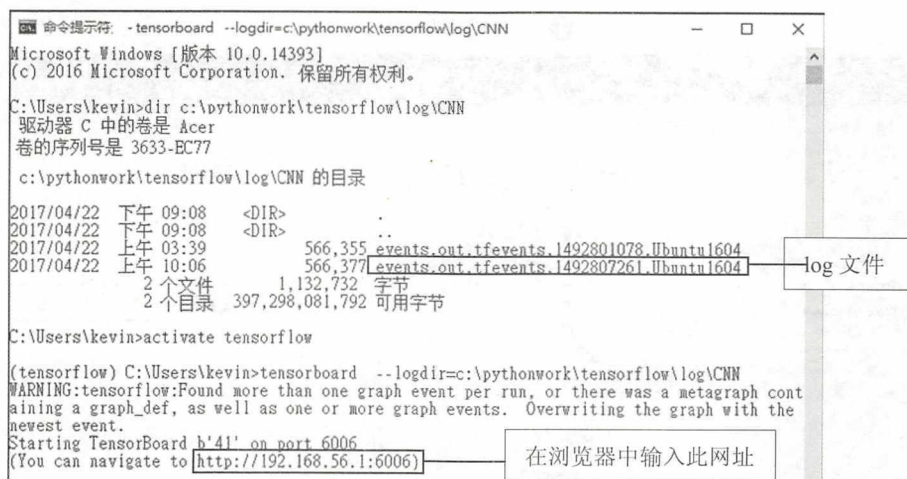


图 19-10

以上执行结果中的 `http://192.168.56.1:6006` 是笔者个人计算机的内部 IP，读者的 IP 可能不相同。读者也可以用浏览器输入此网址：`http://localhost:6006/`，`localhost` 代表本机，就是用户当前使用的计算机。

2. 启动 TensorBoard

启动 TensorBoard 的指令如下，需指定 log 文件目录，TensorBoard 会读取此目录，并显示在 TensorBoard 上。

```
tensorboard --logdir=~/.pywork/tensorFlow/log/CNN
```

执行后屏幕显示界面如下：

```

user@Ubuntu1604: ~
user@Ubuntu1604:~$ tensorboard --logdir=~/.pywork/tensorflow/log/CNN
Starting TensorBoard b'41' on port 6006
(You can navigate to http://127.0.1.1:6006)
  
```

3. 在 TensorBoard 查看“计算图”

启动 TensorBoard 之后，再启动浏览器，并输入网址：`http://localhost:6006/`。

输入网址后就会出现 TensorBoard 界面，在菜单中选择 GRAPHS 之后即可看到“计算图”，如图 19-11 所示。

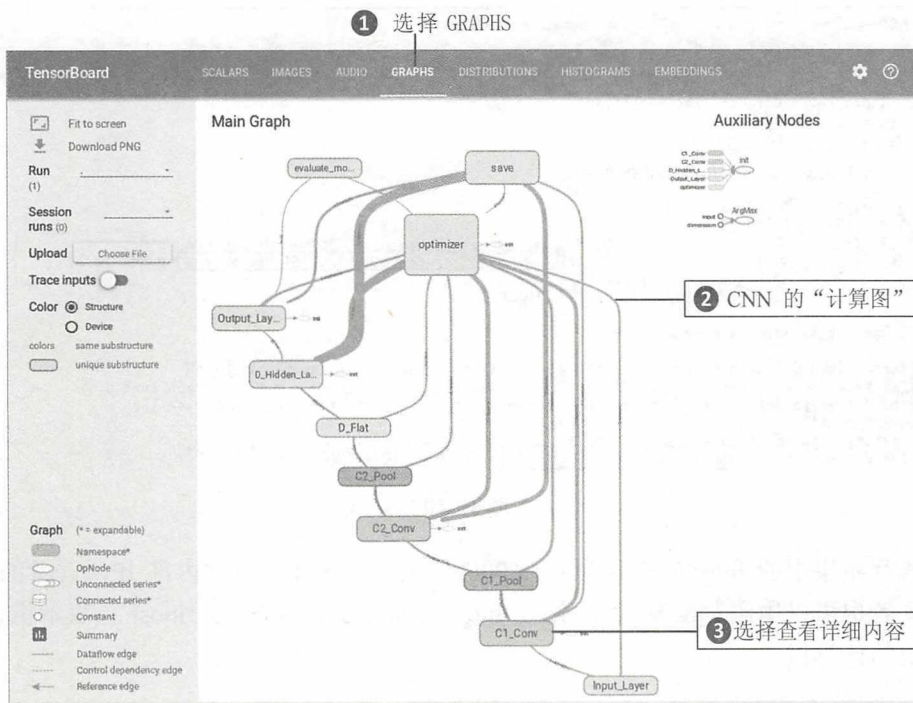


图 19-11

从图 19-11 中可以清楚地看到卷积神经网络的每一个层级，可以单击相应层级来查看详细的内容。

19.11 结论

我们使用卷积神经网络来识别 MNIST 数据集中的手写数字，其分类精度接近 0.99。不过，卷积神经网络训练需要很多时间，下一章我们将介绍如何使用 GPU 来进行训练，这样可以减少训练所需的时间。

第20章

TensorFlow GPU版本的安装

近年来深度学习和人工智能技术发展持续加速，很重要的因素是 GPU 提供了强大的并行计算架构，可让深度学习的训练比普通 CPU 快数十倍。本章将特别介绍 GPU 的安装与应用，读者只需要有 NVIDIA 显示适配器（即显卡），然后安装 CUDA、cuDNN、TensorFlow GPU 版本与 Keras，就可以使用 GPU 大幅加快深度学习的训练。

TensorFlow 主要是通过 NVIDIA 提供的 CUDA 和 cuDNN 来存取 GPU 的，而 Keras 是 TensorFlow 的高级 API，所以必须通过 TensorFlow 存取 GPU，整理如图 20-1 所示。

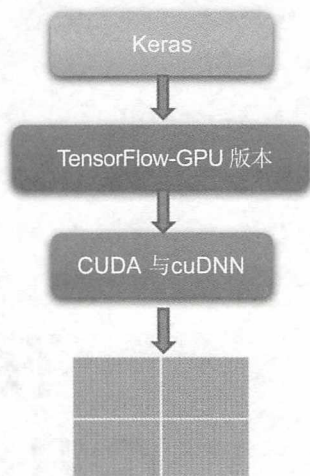


图 20-1 GPU 具有数千个核心

CUDA 是由 NVIDIA 所推出的整合技术，统一计算设备架构（Compute Unified Device Architecture, CUDA），是 NVIDIA 的通用并行计算架构，就是运用图形处理单元（GPU）的强大处理能力大幅增加计算性能。NVIDIA 已售出数百万颗 CUDA GPU，应用于各种领域，如图像处理、视频处理、医学诊断等。

cuDNN（CUDA Deep Neural Network Library）是 NVIDIA 深度学习 SDK 的一部分，是 GPU 的深度学习程序库。cuDNN 能为深度学习提供高性能神经网络层级，例如卷积、池化和激活层等。

我们可以从 NVIDIA 官方网站下载 CUDA 和 cuDNN 这两个软件。

Keras 与 TensorFlow GPU 在 Windows 系统中的安装步骤如图 20-2 所示。



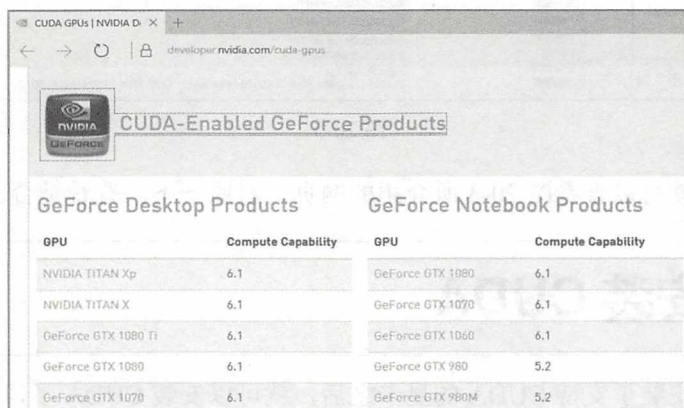
图 20-2

20.1 确认显卡是否支持 CUDA

确认现有的或预定要采购的独立显卡是否支持 CUDA，可按照下列步骤操作。

1. 查看支持 CUDA 的显卡

可到网站 <https://developer.nvidia.com/cuda-gpus> 查看支持 CUDA 的显卡，如图 20-3 所示。



The screenshot shows the NVIDIA Developer website page titled "CUDA-Enabled GeForce Products". It lists supported GPUs for Desktop and Notebook products, categorized by Compute Capability.

GeForce Desktop Products		GeForce Notebook Products	
GPU	Compute Capability	GPU	Compute Capability
NVIDIA TITAN Xp	6.1	GeForce GTX 1080	6.1
NVIDIA TITAN X	6.1	GeForce GTX 1070	6.1
GeForce GTX 1080 Ti	6.1	GeForce GTX 1060	6.1
GeForce GTX 1080	6.1	GeForce GTX 980	5.2
GeForce GTX 1070	6.1	GeForce GTX 980M	5.2

图 20-3

2. 查看系统信息（见图 20-4）

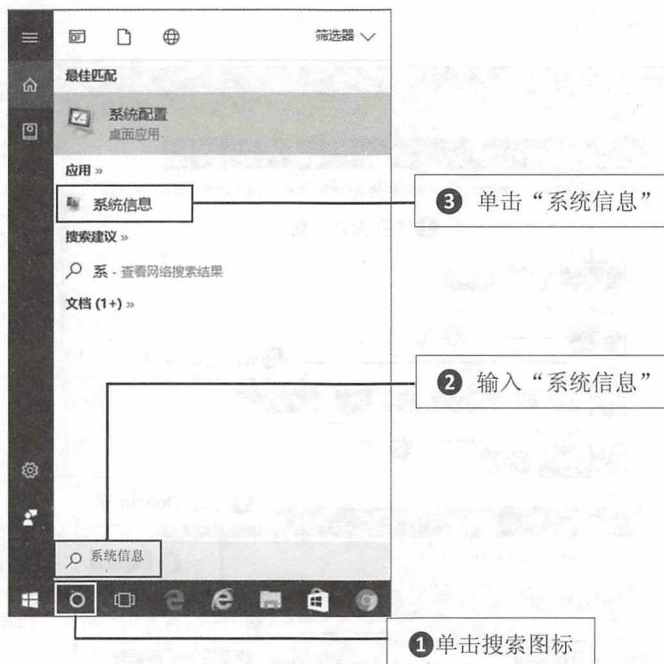


图 20-4

3. 查看显卡（见图 20-5）

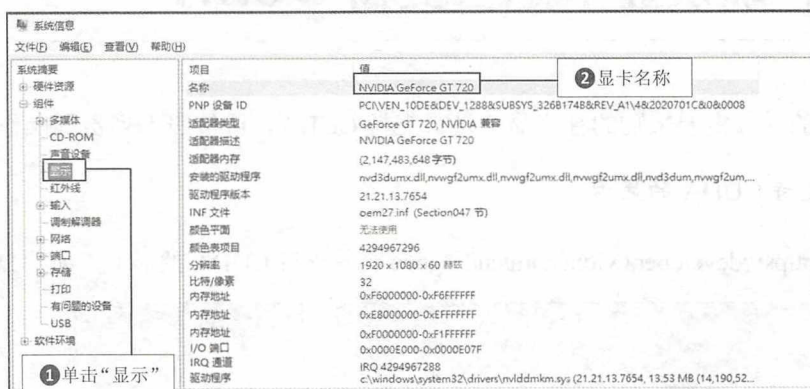


图 20-5

有了显卡名称就可以查看图 20-3 所介绍的网页，对照一下，看看是否支持 CUDA。

20.2 安装 CUDA

确认系统已经安装了支持 CUDA 的显卡之后，就可以安装 CUDA 了，步骤如下。

步骤01 下载并安装 CUDA

到 NVIDIA 网站下载 CUDA: <https://developer.nvidia.com/cuda-downloads>。

安装步骤如图 20-6 所示。

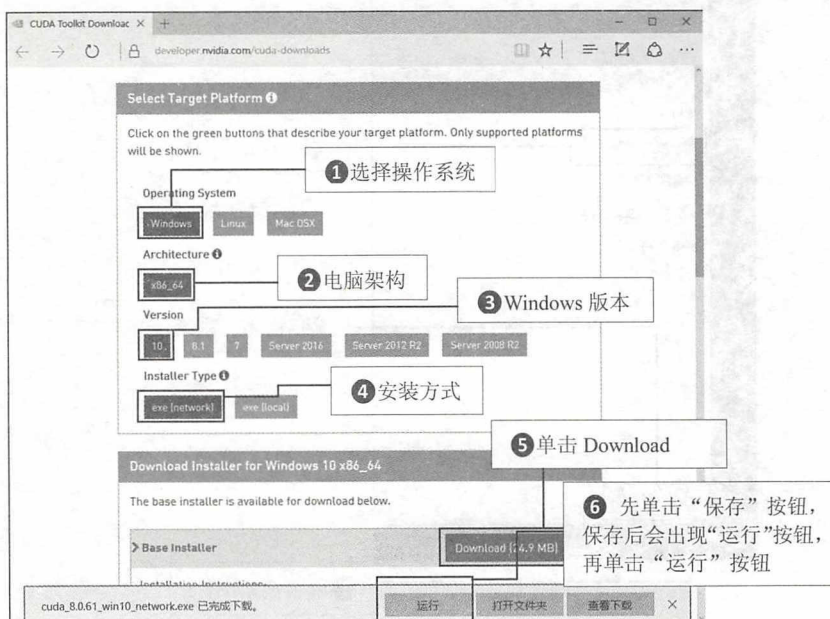


图 20-6



Installer Type 安装有以下两种方式可选。

- **exe(network)**: 下载时文件比较小, 后续执行安装时再下载其余部分。
- **exe(local)**: 下载时完整下载, 后续执行安装时就不需要下载了。

步骤02 设置解压缩安装程序的暂存目录。

因为之前在 Installer Type 中选择了 exe(network), 所以安装过程需要下载与解压缩, 必须设置解压缩目录, 如图 20-7 所示。

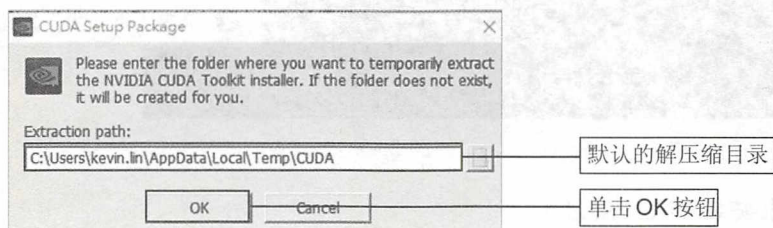


图 20-7

步骤03 同意协议并继续, 如图 20-8 所示。

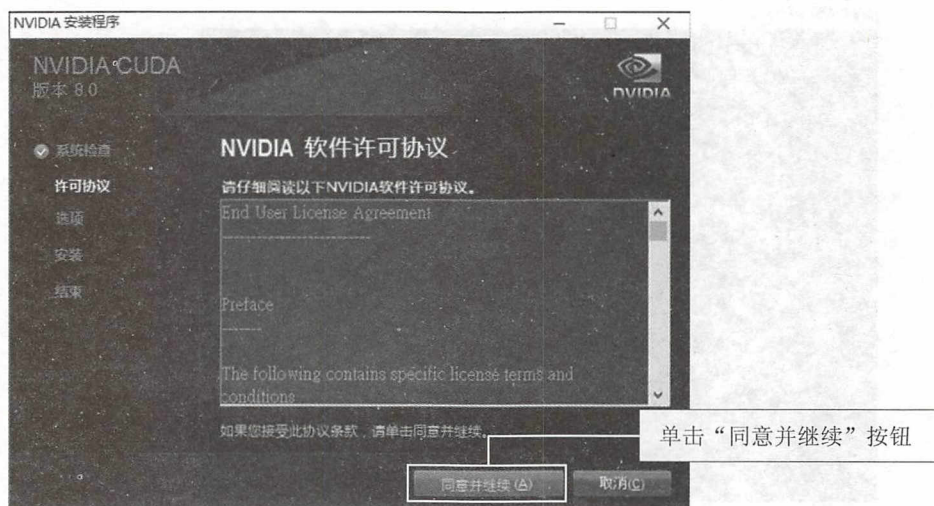


图 20-8

步骤04 选择精简安装选项, 如图 20-9 所示。

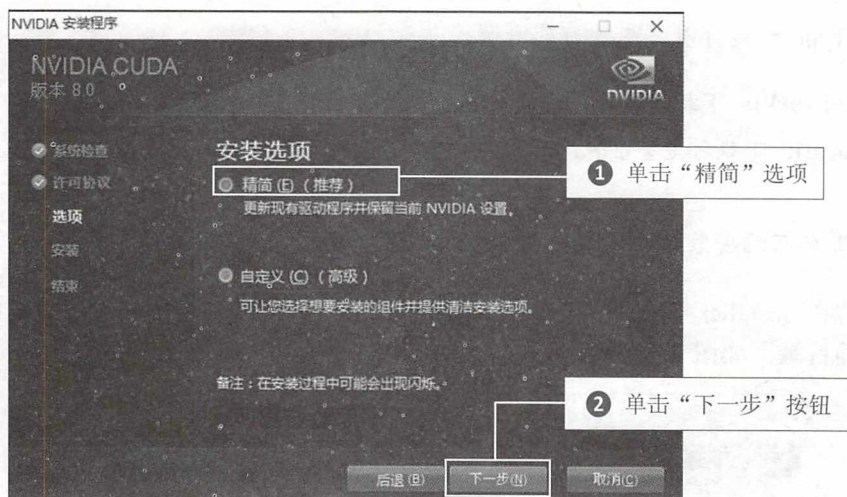


图 20-9

步骤05 警告未安装 Visual Studio。

CUDA 可以用 Visual Studio 开发。图 20-10 所示的界面警告未安装 Visual Studio。不过后续我们是使用 Python 来开发的，所以不需要事先安装 Visual Studio。

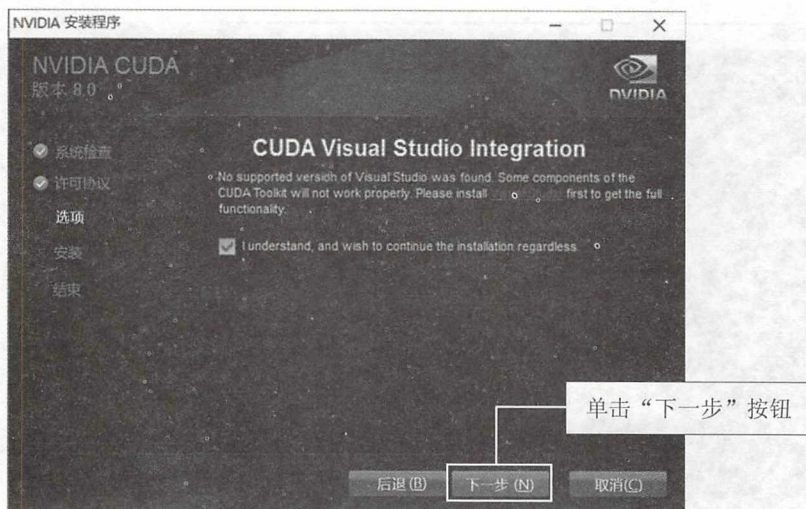


图 20-10

步骤06 下载 CUDA 界面。

因为之前在 Installer Type 中选择了 exe(network)，所以下载会比较久一些，下载过程如图 20-11 所示。



图 20-11

步骤07 确认是否要安装此设备软件，如图 20-12 所示。

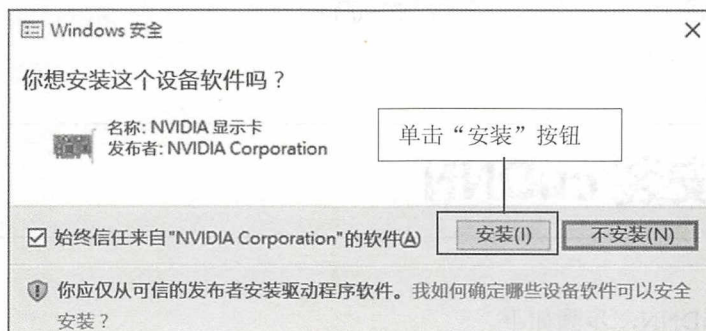


图 20-12

步骤08 确认安装，如图 20-13 所示。



图 20-13

步骤09 安装完成，如图 20-14 所示。



图 20-14

20.3 安装 cuDNN

接下来安装 cuDNN，步骤如下。

步骤01 下载 cuDNN。

在浏览器中输入 NVIDIA 的网址（<https://developer.nvidia.com/cudnn>），出现如图 20-15 所示的页面。

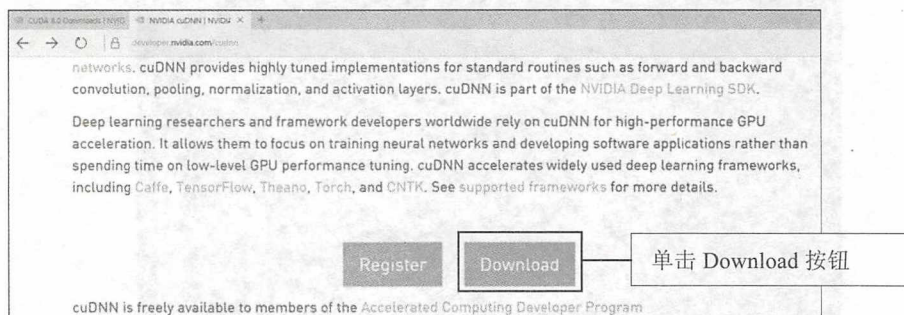


图 20-15

步骤02 加入会员。

下载 cuDNN 必须先成为加速计算开发者计划的会员，如图 20-16 所示。

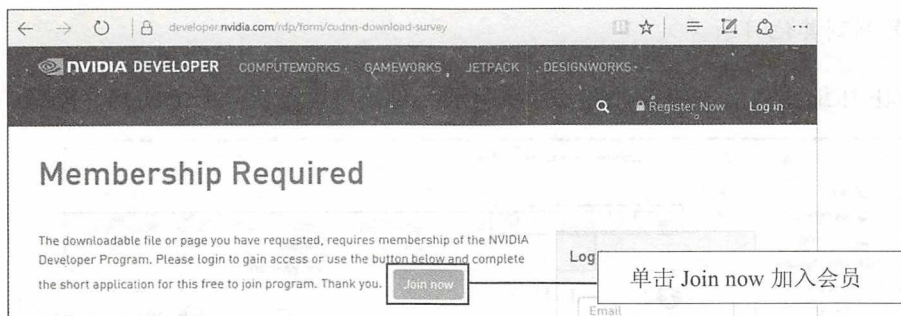


图 20-16

步骤03 进入下载页面。

加入会员并且登录后，进入下载页面，如图 20-17 所示。

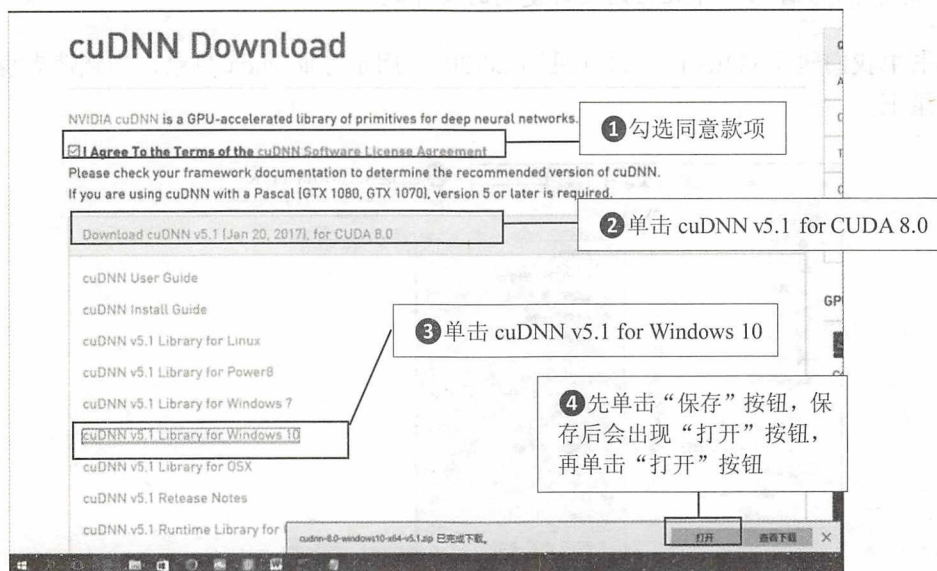


图 20-17

步骤04 查看下载后的文件。

下载后的文件 `cudnn-8.0-windows10-x64-v5.1.zip` 是一个 ZIP 压缩文件，在 Windows 10 中可直接打开，如图 20-18 所示。

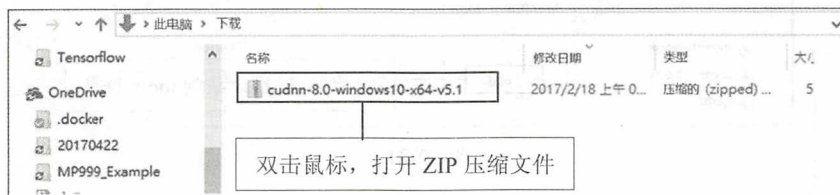


图 20-18

步骤05 复制到其他目录。

打开 ZIP 压缩文件后的内容如图 20-19 所示，我们可以看到一个 CUDA 目录。

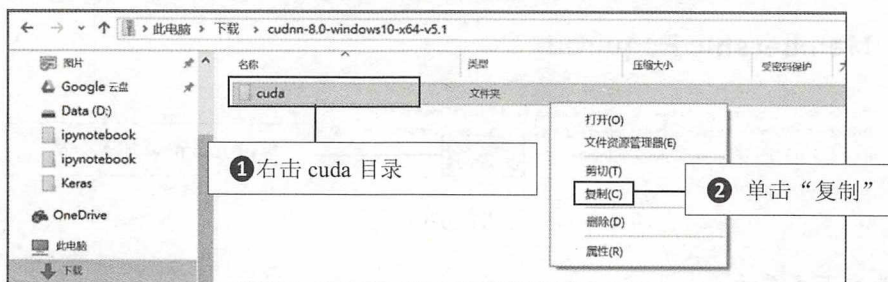


图 20-19

步骤06 创建 tools 目录，并且粘贴之前复制的文件。

在本书中我们建立 C:\tools 目录（见图 20-20），用于存储 cuda 目录，当然读者也可以放在其他目录下。

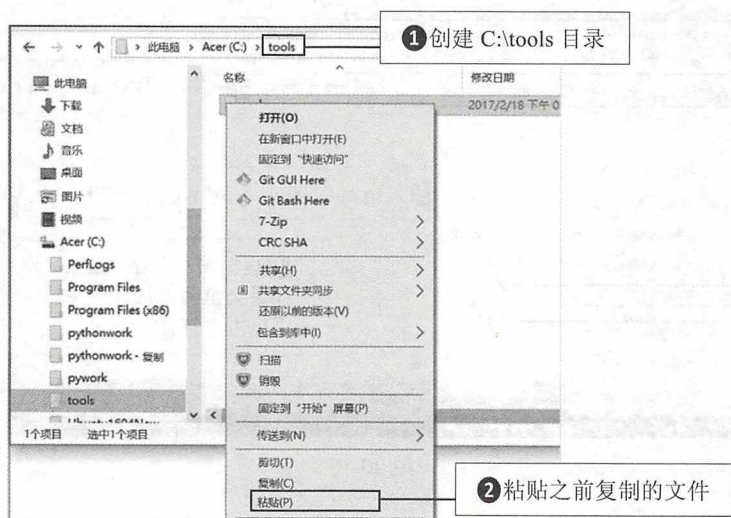


图 20-20

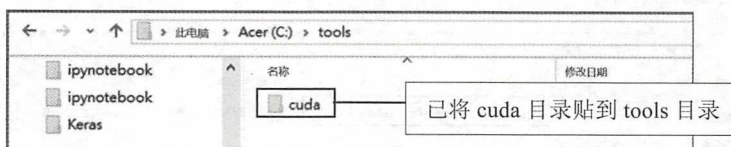
步骤07 已复制/粘贴到 tools 目录，如图 20-21 所示。

图 20-21

步骤08 查看 cudnn64_5.dll。

复制/粘贴后，在 C:\tools\cuda\bin 目录可以看到 cudnn64_5.dll，如图 20-22 所示。这是动

态链接程序库，其他程序会通过此链接库来使用 cuDNN 的功能。

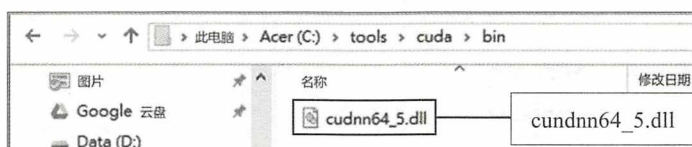


图 20-22

20.4 将 cudnn64_5.dll 存放的位置加入 Path 环境变量

为了让 Windows 系统知道所安装 cuDNN 的目录，必须设置 Path 环境变量，这样其他程序才可以通过这个设置来存取 cudnn64_5.dll。

步骤01 编辑系统环境变量，如图 20-23 所示。

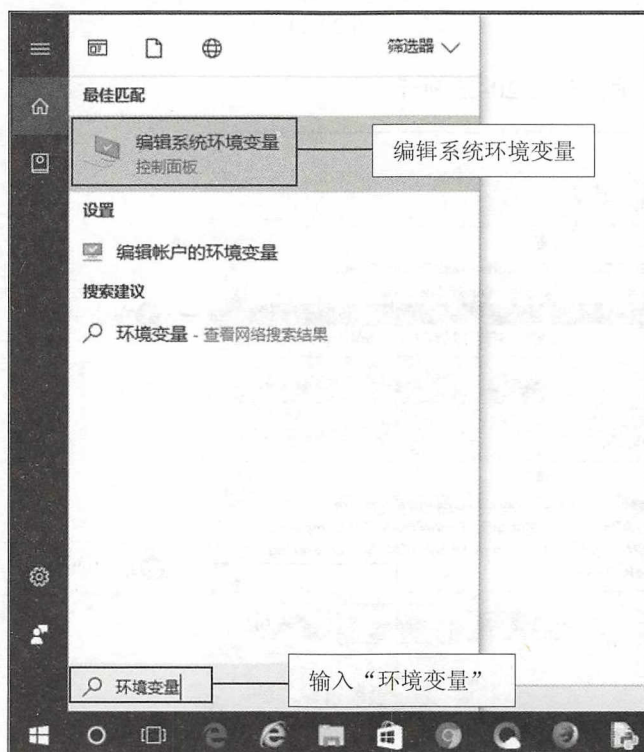


图 20-23

步骤02 单击“环境变量”按钮，如图 20-24 所示。

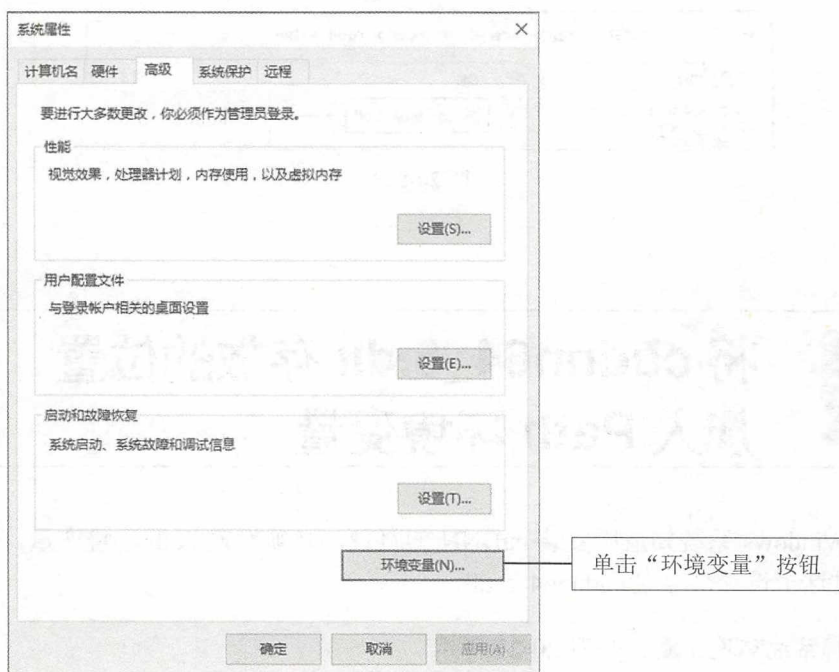


图 20-24

步骤03 编辑环境变量，如图 20-25 所示。

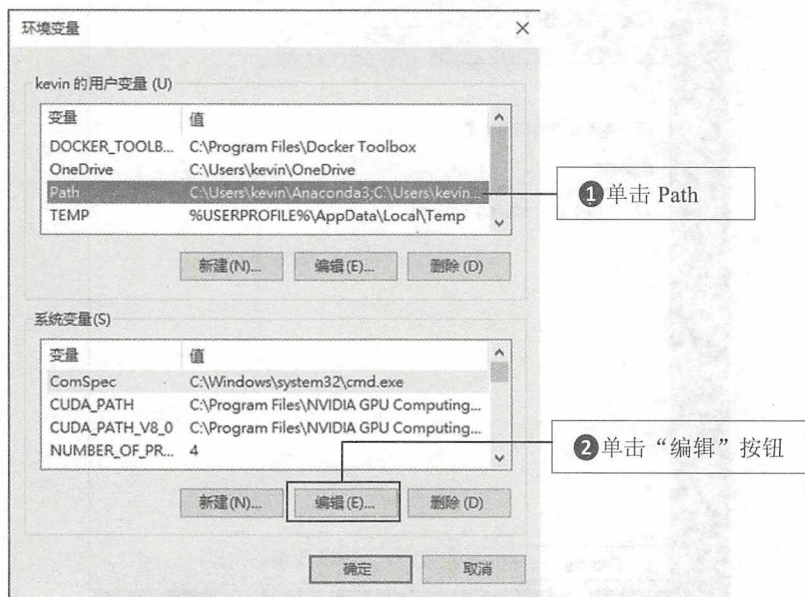


图 20-25

步骤04 编辑用户变量。

在 Path 环境变量中加入“C:\tools\cuda\bin;”，这是 20.3 节 cudnn64_5.dll 的安装目录，如



图 20-26 所示。

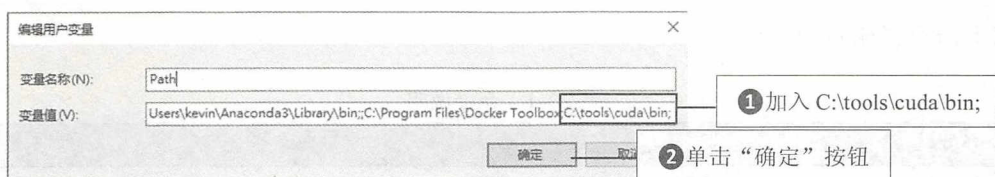


图 20-26

20.5 在 Anaconda 建立 TensorFlow GPU 虚拟环境

本书前面介绍了如何使用 CPU 与 GPU 来执行 TensorFlow 与 Keras。然而 CPU 与 GPU 所需安装的 TensorFlow 版本不一样，所以我们要分别建立 CPU 与 GPU 的虚拟环境，以便在下一章测试 CPU 与 GPU 的执行性能。

在第 4 章中，我们已经介绍了如何在 Anaconda 中建立 TensorFlow 虚拟环境，并且安装了 TensorFlow 的 CPU 版本，本章将在 Anaconda 中建立 TensorFlow GPU 虚拟环境，并且安装 TensorFlow GPU 版本以及 Keras。如果读者还没有在 Windows 中安装 Anaconda，可先按照第 4 章的说明安装 Anaconda。

步骤01 重新启动“命令提示符”程序，并且切换到工作目录。

在“命令提示符”窗口中输入下列命令，以切换到工作目录：

```
cd \pythonwork
```

执行后屏幕显示界面如图 20-27 所示。

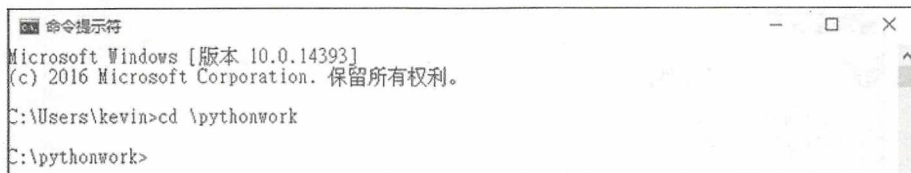


图 20-27

步骤02 在 Anaconda 建立 TensorFlow GPU 虚拟环境。

现在我们要在 Anaconda 建立 TensorFlow GPU 虚拟环境，在此虚拟环境可以安装 TensorFlow GPU 版本，Python 版本是 3.5。


```
conda create --name tensorflow-gpu python=3.5 anaconda
```

以上命令说明见表 20-1。

表 20-1 命令说明

命令	说明
conda create	建立虚拟环境
--name tensorflow-gpu	虚拟环境的名称是 tensorflow-gpu
python=3.5	Python 版本是 3.5
anaconda	加入此命令选项，建立虚拟环境时，也会同时安装其他 Python 软件包，例如 Jupyter Notebook、NumPy、SciPy、Matplotlib、Pandas，用于进行数据分析

执行后屏幕显示界面如图 20-28 所示。

```
命令提示符 - conda create --name tensorflow-gpu python=3.5 anaconda
C:\pythonwork>conda create --name tensorflow-gpu python=3.5 anaconda
Fetching package metadata .....
Solving package specifications: .

Package plan for installation in environment C:\Users\kevin\Anaconda3\envs\tensorflow-gpu
:

The following NEW packages will be INSTALLED:

 _license:                  1.1-py35_1
 alabaster:                 0.7.9-py35_0
 anaconda:                 4.3.1-np111py35_0
 anaconda-client:          1.6.0-py35_0
 anaconda-navigator:       1.5.0-py35_0
 anaconda-project:         0.4.1-py35_0
 argcomplete:              1.0.0-py35_1
 xlswriter:                0.9.6-py35_0
 xlwings:                  0.10.2-py35_0
 xlwt:                     1.2.0-py35_0
 zlib:                     1.2.8-vc14_3          [vc14]

Proceed ([y]/n)? y
```

图 20-28

按下 Y 键之后，就会开始安装 Anaconda 虚拟环境，并会安装软件包。
安装完成后的屏幕显示界面如图 20-29 所示。

```
命令提示符
S 复制了 1个文件。
#
# To activate this environment, use:
# > activate tensorflow
#
# To deactivate this environment, use:
# > deactivate tensorflow
#
# * for power-users using bash, you must source
#
```

图 20-29

步骤03 启用 TensorFlow GPU 虚拟环境。

在“命令提示符”窗口输入下列命令。

➤ 启动 Anaconda 虚拟环境

```
activate tensorflow-gpu
```

执行后屏幕显示界面如图 20-30 所示。

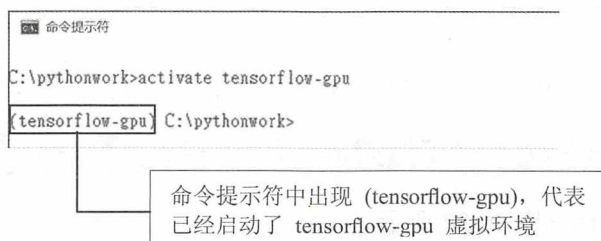


图 20-30

20.6 安装 TensorFlow GPU 版本

接下来，在 TensorFlow GPU 虚拟环境中安装 TensorFlow GPU 版本。
在“命令提示符”窗口中输入下列命令。

➤ 安装 TensorFlow CPU 版本

```
pip install tensorflow-gpu
```

执行后屏幕显示界面如图 20-31 所示。

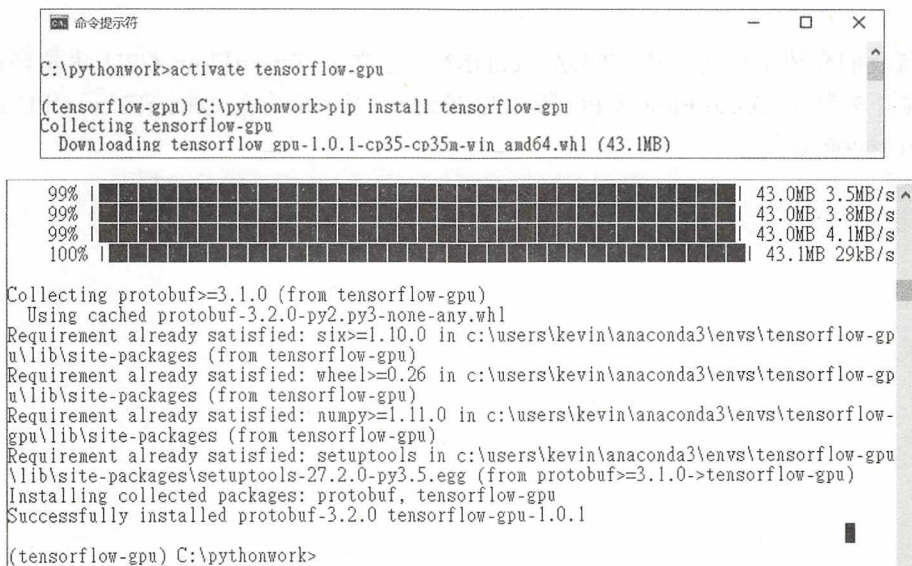


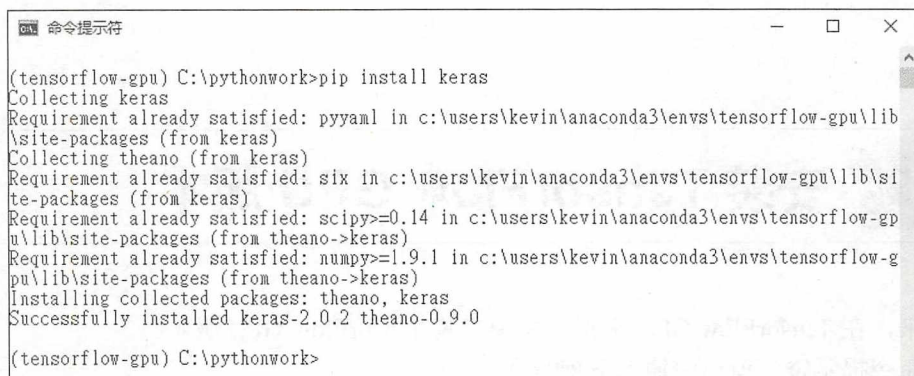
图 20-31

20.7 安装 Keras

在“命令提示符”窗口中输入下列命令来安装 Keras:

```
pip install keras
```

执行后屏幕显示界面如图 20-32 所示。



```
(tensorflow-gpu) C:\pythonwork>pip install keras
Collecting keras
Requirement already satisfied: pyyaml in c:\users\kevin\anaconda3\envs\tensorflow-gpu\lib\site-packages (from keras)
Collecting theano (from keras)
Requirement already satisfied: six in c:\users\kevin\anaconda3\envs\tensorflow-gpu\lib\site-packages (from theano->keras)
Requirement already satisfied: scipy>=0.14 in c:\users\kevin\anaconda3\envs\tensorflow-gpu\lib\site-packages (from theano->keras)
Requirement already satisfied: numpy>=1.9.1 in c:\users\kevin\anaconda3\envs\tensorflow-gpu\lib\site-packages (from theano->keras)
Installing collected packages: theano, keras
Successfully installed keras-2.0.2 theano-0.9.0
(tensorflow-gpu) C:\pythonwork>
```

图 20-32

20.8 结论

在本章我们介绍了如何安装 CUDA、cuDNN，建立了 TensorFlow GPU 虚拟环境，并且在虚拟环境下安装了 TensorFlow GPU 版本与 Keras，下一章将在 TensorFlow GPU 虚拟环境中测试 GPU 的强大功能。

第21章

使用GPU加快TensorFlow 与Keras训练

本章我们将在 TensorFlow GPU 虚拟环境中测试 GPU 的强大功能，并且分别测试 TensorFlow 与 Keras 在 CPU 与 GPU 中执行速度的差异。

如果读者还没有安装 TensorFlow GPU 版本与 Keras, 就按照第 20 章的说明安装。本章完整的程序代码可参考范例程序 Test_GPU.ipynb。本书范例程序的下载与安装说明可参考附录 A。

21.1 启动 TensorFlow GPU 环境

1. 启动 TensorFlow GPU

➤ 切换工作目录

```
cd \pywork
```

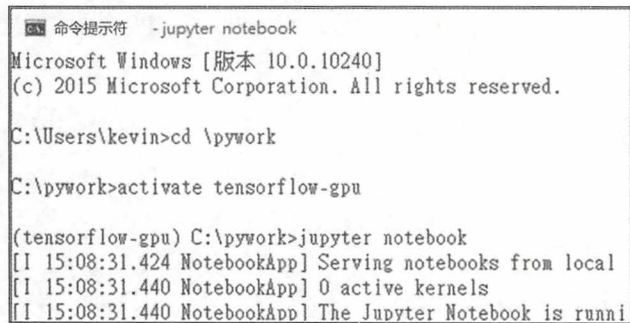
➤ 启动 TensorFlow GPU 环境

```
activate tensorflow-gpu
```

➤ 启动 jupyter notebook

```
jupyter notebook
```

执行后屏幕显示界面如图 21-1 所示。



```
命令提示符 - jupyter notebook
Microsoft Windows [版本 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\kevin>cd \pywork

C:\pywork>activate tensorflow-gpu

(tensorflow-gpu) C:\pywork>jupyter notebook
[I 15:08:31.424 NotebookApp] Serving notebooks from local
[I 15:08:31.440 NotebookApp] 0 active kernels
[I 15:08:31.440 NotebookApp] The Jupyter Notebook is runni
```

图 21-1

执行后开启 Jupyter, 如图 21-2 所示。

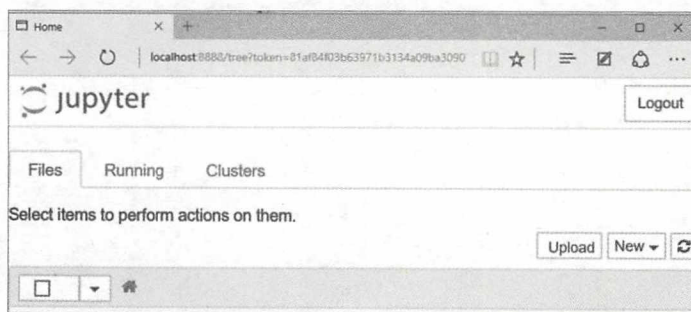


图 21-2

参考第 4 章的说明建立新的 Notebook。

2. 导入 tensorflow 模块

在 jupyter notebook 中输入下列程序代码。

➤ 导入 tensorflow 模块

```
In [1]: import tensorflow as tf
import time
```

以上导入 tensorflow 模块的命令执行后，在“命令提示符”中的 jupyter notebook 界面可以同步看到 TensorFlow 自动打开了很多 dll 文件，这些 dll 是 CUDA 链接库，如图 21-3 所示。如果出现加载 dll 的错误信息，就代表之前在 20.4 节中设置的环境变量可能有错误。

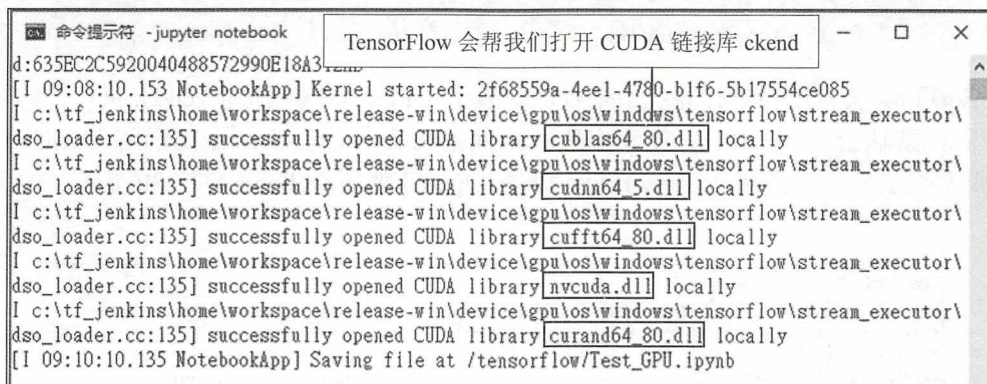


图 21-3

3. 默认以 GPU 执行

➤ 建立计算图

下面的程序主要是建立两个矩阵：W 与 X，矩阵的大小由 size 变量来设置，目前设置为 size=500，然后进行矩阵乘法 matmul 与矩阵元素的加总。

```
In [2]: size=500
W = tf.random_normal([size, size],name='W')
X = tf.random_normal [size, size],name='X')
mul = tf.matmul(W, X,name='mul')
sum_result = tf.reduce_sum(mul,name='sum')

with tf.Session() as sess:
    result = sess.run(sum_result)
    print('result=',result)

result= -593.626
```

程序代码的说明见表 21-1。

表 21-1 程序代码说明

程序代码	说明
size=500	设置 size 为 500
W = tf.random_normal([size, size],name='W')	以随机产生数值方式建立 W 矩阵, shape 为[size, size], 也就是[500, 500]
X = tf.random_normal([size, size],name='X')	以随机产生数值方式建立 X 矩阵, shape 为[size, size], 也就是[500,500]
mul = tf.matmul(W, X,name='mul')	使用 tf.matmul 将 W 与 X 矩阵进行矩阵乘法运算, 结果存放在 mul 矩阵中
sum_result = tf.reduce_sum(mul,name='sum')	使用 tf.reduce_sum 将 mul 矩阵内的值加总

➤ 执行计算图

```
with tf.Session() as sess:
    result = sess.run(sum_result)
```

TensorFlow 会自动检查计算机中是否已安装了 GPU, 因为理论上 GPU 执行得比较快, TensorFlow 默认会使用 GPU。当我们执行上面的命令时, 在“命令提示符”中的 jupyter notebook 界面会同步看到建立 TensorFlow 设备, /gpu:0 使用 GeForce GT 720 显卡, 如图 21-4 所示。

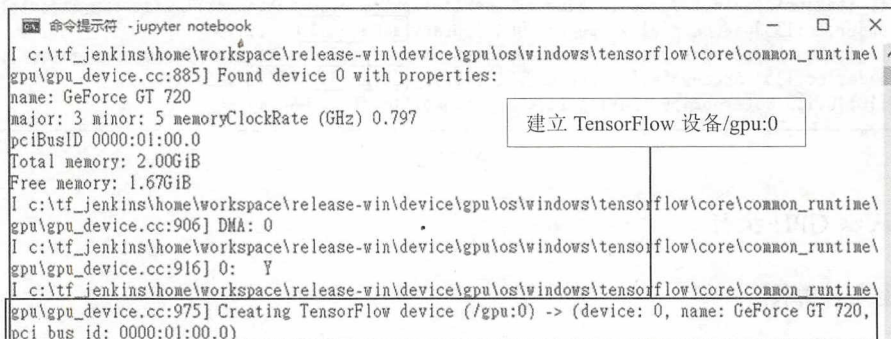


图 21-4

4. 设置 Session 的 config 参数, 显示更多 GPU 设备信息

在之前的步骤中显示的 GPU 设备相关信息比较少, 我们可以用下列指令来显示更多设备相关信息。

```
In [3]: size=500
W = tf.random_normal([size, size],name='W')
X = tf.random_normal([size, size],name='X')
mul = tf.matmul(W, X,name='mul')
sum_result = tf.reduce_sum(mul,name='sum')

tfconfig=tf.ConfigProto(log_device_placement=True)
with tf.Session(config=tfconfig) as sess:
    result = sess.run(sum_result)
```

程序代码的说明如下:

➤ 建立 Session 的配置设置

使用 `tf.ConfigProto` 建立 session 的配置设置 `tfconfig`，传入参数 `log_device_placement` 设置为 `True`，代表要显示设备的相关信息。

```
tfconfig=tf.ConfigProto(log_device_placement=True)
```

➤ 打开 Session 时传入配置设置

打开 Session 时传入之前建立的 Session 的配置 `tfconfig`：

```
with tf.Session(config=tfconfig) as sess:
```

当我们执行以上指令时，在“命令提示符”中的 jupyter notebook 界面会同步显示使用设备 `/gpu:0` 进行计算，如图 21-5 所示。



图 21-5

5. 以 `with tf.device` 来指定使用 CPU 或 GPU 设备

之前的步骤 TensorFlow 默认使用 `/gpu:0`。但是在某些情况下，我们希望能指定使用的计算设备，此时可以使用 `with tf.device` 语句来指定某一段程序使用哪一个计算设备，例如：

- 计算机中有多个 GPU，我们可以指定要使用哪一个具体的 GPU。

- `with tf.device("/gpu:0")`：使用第 0 个 GPU。

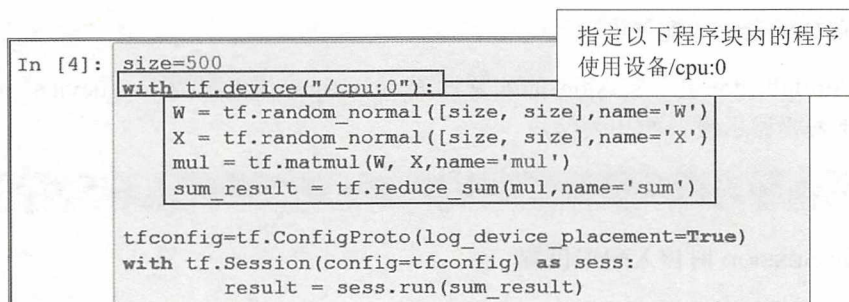
- `with tf.device("/gpu:1")`：使用第 1 个 GPU。

- 指定用 CPU 来执行。

- `with tf.device("/cpu:0")`：使用第 0 个 CPU。

6. 以 `with tf.device` 来指定使用 CPU

指定用 CPU 进行计算的程序代码。



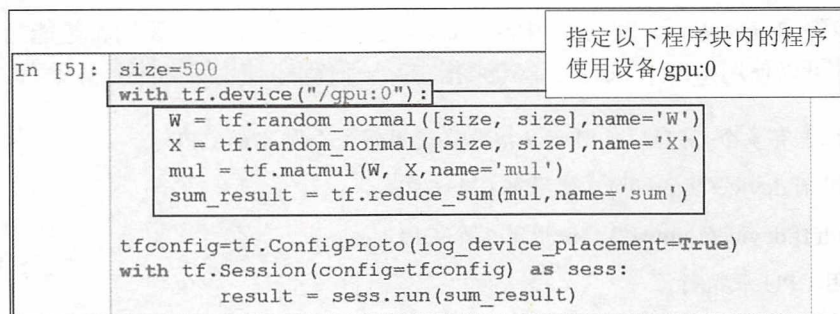
以上命令执行后，在“命令提示符”中的 jupyter notebook 界面会同步显示使用设备 /cpu:0 进行计算，如图 21-6 所示。



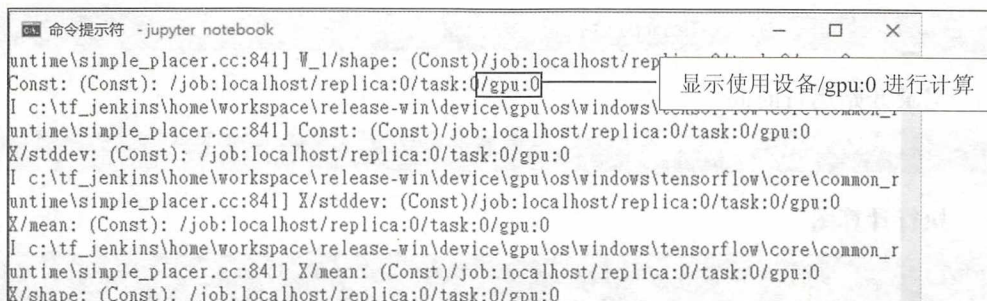
图 21-6

7. 以 with tf.device 来指定使用 GPU

指定以 GPU 进行计算的程序代码。



以上命令执行后，在“命令提示符”中的 jupyter notebook 界面会同步显示使用设备 /gpu:0 进行计算，如图 21-7 所示。



```

untimesimple_placer.cc:841] W_1/shape: (Const)/job:localhost/rep
Const: (Const): /job:localhost/replica:0/task:0/gpu:0
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\common_r
untimesimple_placer.cc:841] Const: (Const)/job:localhost/replica:0/task:0/gpu:0
X/stddev: (Const): /job:localhost/replica:0/task:0/gpu:0
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\common_r
untimesimple_placer.cc:841] X/stddev: (Const)/job:localhost/replica:0/task:0/gpu:0
X/mean: (Const): /job:localhost/replica:0/task:0/gpu:0
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\common_r
untimesimple_placer.cc:841] X/mean: (Const)/job:localhost/replica:0/task:0/gpu:0
X/shape: (Const): /job:localhost/replica:0/task:0/gpu:0
  
```

图 21-7

21.2 测试 GPU 与 CPU 执行性能

我们将创建 performanceTest，用于测试 GPU 与 CPU 的执行性能。

1. 创建 performanceTest 函数

```

In [6]: def performanceTest(device_name,size):
        with tf.device(device_name):
            W = tf.random_normal([size, size],name='W')
            X = tf.random_normal([size, size],name='X')
            mul = tf.matmul(W, X,name='mul')
            sum_result = tf.reduce_sum(mul,name='sum')

            startTime = time.time()
            tfconfig=tf.ConfigProto(log_device_placement=True)
            with tf.Session(config=tfconfig) as sess:
                result = sess.run(sum_result)
            takeTimes=time.time() - startTime
            print(device_name," size=",size,"Time:",takeTimes )
        return takeTimes
  
```

程序代码的说明如下：

➤ 定义 performanceTest 函数需传入参数：

- **device_name** 设置要执行计算的设备，例如 /cpu:0 或 /gpu:0。
- **size** 设置要建立矩阵的大小。

```
def performanceTest(device_name,size):
```

➤ 使用 with tf.device 语句指定要使用的计算设备

使用 with tf.device 传入 device_name 参数，指定下列程序块中的程序代码要使用的计算设备。

```

with tf.device(device_name):
    W = tf.random_normal([size, size],name='W') X = tf.random_normal([size,
size],name='X') mul = tf.matmul(W, X,name='mul')
  
```

```
sum_result = tf.reduce_sum(mul, name='sum')
```

➤ 记录开始运行时间

```
startTime = time.time()
```

➤ 执行计算图

```
tfconfig=tf.ConfigProto(log_device_placement=True) with tf.Session(config=
tfconfig) as sess:
    result = sess.run(sum_result)
```

➤ 计算运行时间

将当前的时间减去开始运行时间就是执行所需的时间。

```
takeTimes=time.time() - startTime print(device_name," size=",size,"Time:",
takeTimes )
```

➤ 返回运行时间

```
return takeTimes
```

2. 执行 performanceTest

以下程序代码执行 performanceTest 分别传入：

- g=performanceTest("/gpu:0",100) 用 GPU 执行计算，建立的矩阵大小为 100。
- c=performanceTest("/cpu:0",100) 用 CPU 执行计算，建立的矩阵大小为 100。

```
In [7]: g=performanceTest("/gpu:0",100)
        c=performanceTest("/cpu:0",100)

/gpu:0 size= 100 Time: 0.06372833251953125
/cpu:0 size= 100 Time: 0.08257627487182617
```

我们可以看到以上执行结果 GPU 大约 0.06 秒，CPU 大约 0.08 秒，GPU 比较快。

3. 创建 performanceTest

以下程序代码使用 for 循环执行 performanceTest，使用不同的设备并设置产生矩阵的大小。

```
In [11]: gpu_set=[];cpu_set=[];i_set=[]
        for i in range(0,5001,500):
            g=performanceTest("/gpu:0",i)
            c=performanceTest("/cpu:0",i)
            gpu_set.append(g);cpu_set.append(c);i_set.append(i)
```

程序代码的说明如下：



➤ 初始化 gpu_set、cpu_set、i_set

我们将使用 gpu_set、cpu_set、i_set 记录 GPU 与 CPU 的运行时间。

```
gpu_set=[];cpu_set=[];i_set=[]
```

➤ 使用 for 循环重复执行程序块内的程序代码

使用 for 循环重复执行程序块内的程序代码，其中 range(0,5001,500)会产生序列：从 0 到 5001 间隔 500，也就是说会产生序列[0, 500, 1000, ..., 5000]，每次设置 i 值为产生的序列。分别用 GPU 及 CPU 执行 performanceTest，并把矩阵大小设置为 i。

```
for i in range(0,5001,500): g=performanceTest("/gpu:0",i) c=performanceTest("/cpu:0",i)
```

➤ 将每次的执行结果存储在 gpu_set、cpu_set、i_set 中

```
gpu_set.append(g);cpu_set.append(c);i_set.append(i)
```

执行后屏幕显示界面如图 21-8 所示。

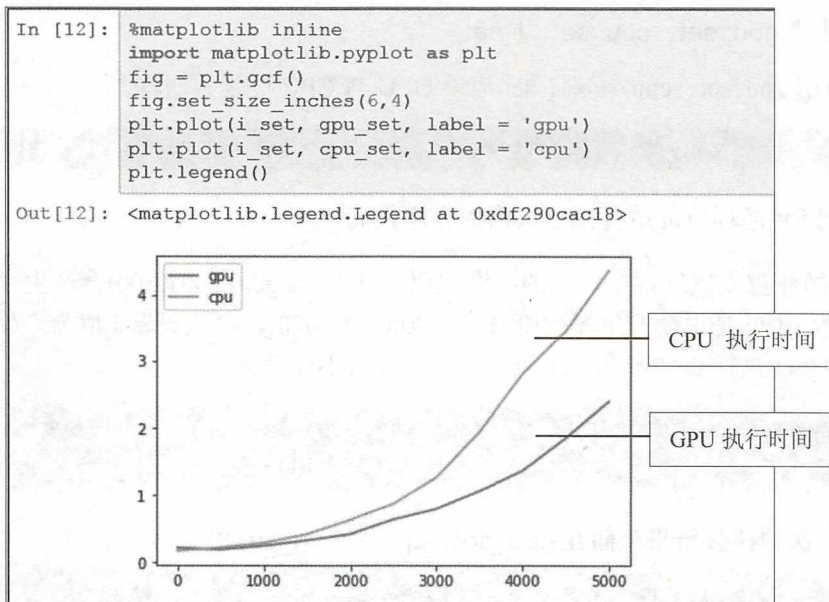
```
/gpu:0 size= 0 Time: 0.22510099411010742
/cpu:0 size= 0 Time: 0.18112730979919434
/gpu:0 size= 500 Time: 0.1986405849456787
/cpu:0 size= 500 Time: 0.22916150093078613
/gpu:0 size= 1000 Time: 0.2545931339263916
/cpu:0 size= 1000 Time: 0.2967100143432617
/gpu:0 size= 1500 Time: 0.3357372283935547
/cpu:0 size= 1500 Time: 0.4268019199371338
/gpu:0 size= 2000 Time: 0.4277958869934082
/cpu:0 size= 2000 Time: 0.6374521255493164
/gpu:0 size= 2500 Time: 0.6419293880462646
/cpu:0 size= 2500 Time: 0.873967170715332
/gpu:0 size= 3000 Time: 0.8025901317596436
/cpu:0 size= 3000 Time: 1.275038242340088
/gpu:0 size= 3500 Time: 1.070591688156128
/cpu:0 size= 3500 Time: 1.9502127170562744
/gpu:0 size= 4000 Time: 1.3737995624542236
```

图 21-8

从以上执行结果可以看到 size 是产生矩阵的大小，产生的矩阵越大，所需时间越多。矩阵越大，CPU 与 GPU 的差距越大，也就是说矩阵越大，越能发挥使用 GPU 的性能。

4. 将 GPU 与 CPU 的运行时间用图形来查看

用下列程序代码以图形来显示 GPU 与 CPU 的运行时间，这样更容易看出 GPU 与 CPU 执行速度的差异。



x 轴是矩阵大小, y 轴是运行时间, 矩阵越大, CPU 与 GPU 的差距越大。也就是说, 矩阵越大, 越能发挥使用 GPU 的性能。

21.3 超出显卡内存的限制

笔者所测试的 GeForce GT 720 显卡经过测试发现矩阵大小超过 6000, 由于显卡内存不足, 因此会发生错误。当然, 如果显卡内存比较大, 就可能较大的矩阵才会发生错误。

```
In [*]: g=performanceTest("/gpu:0",7000)
```

出现 NVIDIA 显示驱动程序停止响应, 出现错误时界面如图 21-9 所示。



图 21-9

接下来会显示 Python 已经停止运行, 如图 21-10 所示。

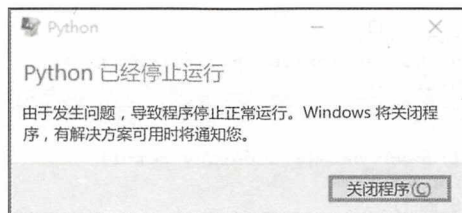


图 21-10

遇到这种情况，可关闭再重新打开“命令提示符”窗口，然后按照 21.1 节的步骤重新启动 TensorFlow GPU 环境。

由以上测试得知，当我们选购显卡时，显卡内存也是选购考虑的重点，内存越大越好，当然成本也更高。

21.4 以多层感知器的实际范例比较 CPU 与 GPU 的执行速度

接下来将以实际范例来比较 CPU 与 GPU 的执行速度，我们将之前第 18 章所创建的 TensorFlow_Mnist_MLP_h1000.ipynb 作为测试范例。

步骤01 启动 TensorFlow CPU 环境。

我们将使用第 3 章所建立的 Anaconda TensorFlow CPU 虚拟环境。

➤ 启动 TensorFlow 环境并启动 jupyter notebook

```
cd \pywork activate tensorflow jupyter notebook
```

执行后屏幕显示界面如图 21-11 所示。

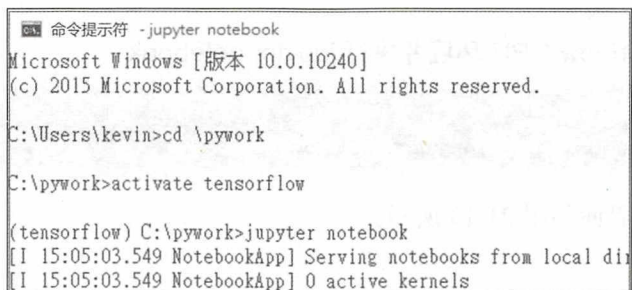


图 21-11

步骤02 在 TensorFlow CPU 环境中执行训练的程序代码及其执行结果如图 21-12 所示。

```

In [10]: sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
trainEpochs = 15
batchSize = 100
loss_set=[];epoch_set=[];accuracy_set=[]
from time import time
startTime=time()
for epoch in range(trainEpochs):
    avgLoss = 0.
    totalBatches = int(mnist.train.num_examples/batchSize)
    for i in range(totalBatches):
        batch_x, batch_y = mnist.train.next_batch(batchSize)
        _, loss = sess.run([optimizer, loss],
                           feed_dict={x: batch_x,y_label: batch_y})
        avgLoss += loss / totalBatches
    acc = sess.run(accuracy,feed_dict={x: mnist.validation.images,
                                       y_label: mnist.validation.labels})
    loss_set.append(avgLoss);epoch_set.append(epoch);
    accuracy_set.append(acc)
    print("Epoch:", '%04d' % (epoch+1), "Loss=", \
          "{:.9f}".format(avgLoss),"Training Accuracy=",acc)
duration =time()-startTime
print("Optimization Finished takes:",duration)

```

```

Epoch: 0001 Loss= 17.154506977 Training Accuracy= 0.897
Epoch: 0002 Loss= 5.805346230 Training Accuracy= 0.9266
Epoch: 0003 Loss= 3.773920338 Training Accuracy= 0.9354
Epoch: 0004 Loss= 2.667857746 Training Accuracy= 0.9418
Epoch: 0005 Loss= 2.060674991 Training Accuracy= 0.9424
Epoch: 0006 Loss= 1.585709708 Training Accuracy= 0.9504
Epoch: 0007 Loss= 1.287338891 Training Accuracy= 0.9508
Epoch: 0008 Loss= 1.016832831 Training Accuracy= 0.9486
Epoch: 0009 Loss= 0.830355296 Training Accuracy= 0.952
Epoch: 0010 Loss= 0.661952721 Training Accuracy= 0.9522
Epoch: 0011 Loss= 0.531144379 Training Accuracy= 0.951
Epoch: 0012 Loss= 0.417248771 Training Accuracy= 0.9534
Epoch: 0013 Loss= 0.341606024 Training Accuracy= 0.9546
Epoch: 0014 Loss= 0.269543508 Training Accuracy= 0.9522
Epoch: 0015 Loss= 0.213566572 Training Accuracy= 0.9524
Optimization Finished takes: 106.38361692428589

```

图 21-12

从以上执行结果可知，使用 CPU 执行共需 106 秒。

步骤03 启动 TensorFlow GPU 环境。

➤ 启动 TensorFlow GPU 环境并执行 jupyter notebook

```

cd \pywork
activate tensorflow-gpu jupyter notebook

```

执行后屏幕显示界面如图 21-13 所示。



```

命令提示符 - jupyter notebook
Microsoft Windows [版本 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\kevin>cd \pywork

C:\pywork>activate tensorflow-gpu

(tensorflow-gpu) C:\pywork>jupyter notebook
[I 15:08:31.424 NotebookApp] Serving notebooks from local
[I 15:08:31.440 NotebookApp] 0 active kernels
[I 15:08:31.440 NotebookApp] The Jupyter Notebook is runni

```

图 21-13

步骤04 打开 TensorFlow_Mnist_MLP_h1000.ipynb，如图 21-14 所示。

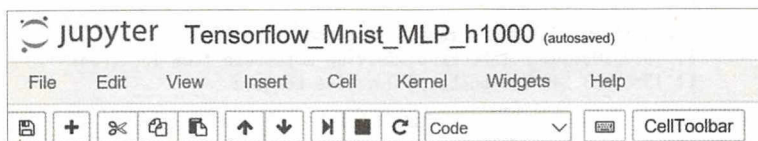


图 21-14

步骤05 执行训练程序代码，如图 21-15 所示。

```

Epoch: 0001 Loss= 16.047079401 Training Accuracy= 0.9116
Epoch: 0002 Loss= 5.659989039 Training Accuracy= 0.9278
Epoch: 0003 Loss= 3.662495367 Training Accuracy= 0.9366
Epoch: 0004 Loss= 2.627829641 Training Accuracy= 0.941
Epoch: 0005 Loss= 1.995857331 Training Accuracy= 0.94
Epoch: 0006 Loss= 1.534073062 Training Accuracy= 0.9476
Epoch: 0007 Loss= 1.205598021 Training Accuracy= 0.947
Epoch: 0008 Loss= 0.947007764 Training Accuracy= 0.9504
Epoch: 0009 Loss= 0.768120100 Training Accuracy= 0.9494
Epoch: 0010 Loss= 0.616772215 Training Accuracy= 0.9522
Epoch: 0011 Loss= 0.483607222 Training Accuracy= 0.9522
Epoch: 0012 Loss= 0.400731914 Training Accuracy= 0.952
Epoch: 0013 Loss= 0.323831484 Training Accuracy= 0.9542
Epoch: 0014 Loss= 0.250744874 Training Accuracy= 0.9536
Epoch: 0015 Loss= 0.207788039 Training Accuracy= 0.9534
Optimization Finished takes: 78.60383820533752

```

图 21-15

以上用 GPU 执行计算共需 78 秒，比起之前使用 CPU 执行运算所需的 106 秒少了 28 秒。大约计算了一下， $(28 \div 106) \times 100 = 26$ ，也就是说 GPU 比 CPU 大约减少 26% 的时间。

21.5

以 CNN 的实际范例比较 CPU 与 GPU 的执行速度

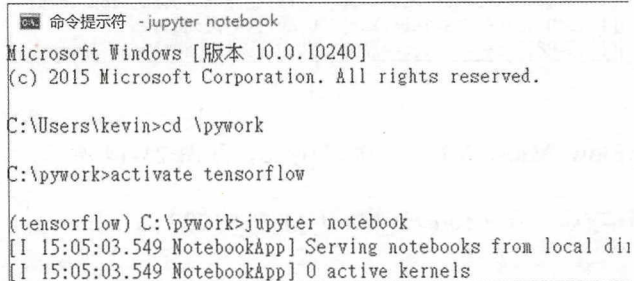
接下来，我们将以实际范例比较 CPU 与 GPU 的执行速度，将之前第 19 章所创建的 TensorFlow_Mnist_CNN.ipynb 作为测试范例。

步骤01 启动 TensorFlow CPU 环境。

➤ 启动 jupyter notebook

```
cd \pywork activate tensorflow jupyter notebook
```

执行后屏幕显示界面如图 21-16 所示。



```
命令提示符 - jupyter notebook
Microsoft Windows [版本 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\kevin>cd \pywork

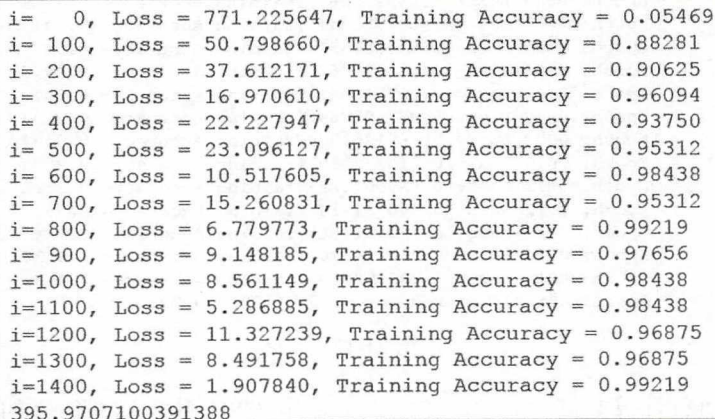
C:\pywork>activate tensorflow

(tensorflow) C:\pywork>jupyter notebook
[I 15:05:03.549 NotebookApp] Serving notebooks from local dir
[I 15:05:03.549 NotebookApp] 0 active kernels
```

图 21-16

步骤02 在 Anaconda TensorFlow CPU 环境执行训练。

进入 jupyter notebook 界面后，打开并执行 TensorFlow_Mnist_CNN.ipynb 程序，执行训练的界面如图 21-17 所示。



```
i= 0, Loss = 771.225647, Training Accuracy = 0.05469
i= 100, Loss = 50.798660, Training Accuracy = 0.88281
i= 200, Loss = 37.612171, Training Accuracy = 0.90625
i= 300, Loss = 16.970610, Training Accuracy = 0.96094
i= 400, Loss = 22.227947, Training Accuracy = 0.93750
i= 500, Loss = 23.096127, Training Accuracy = 0.95312
i= 600, Loss = 10.517605, Training Accuracy = 0.98438
i= 700, Loss = 15.260831, Training Accuracy = 0.95312
i= 800, Loss = 6.779773, Training Accuracy = 0.99219
i= 900, Loss = 9.148185, Training Accuracy = 0.97656
i=1000, Loss = 8.561149, Training Accuracy = 0.98438
i=1100, Loss = 5.286885, Training Accuracy = 0.98438
i=1200, Loss = 11.327239, Training Accuracy = 0.96875
i=1300, Loss = 8.491758, Training Accuracy = 0.96875
i=1400, Loss = 1.907840, Training Accuracy = 0.99219
395.9707100391388
```

图 21-17

从使用 CPU 执行训练后的结果可知，花费时间是 395 秒。

步骤03 启动 TensorFlow GPU 环境。

关闭之前的“命令提示符”窗口，并打开新的“命令提示符”窗口，输入下列命令启动 TensorFlow GPU 环境，之后再启动 jupyter notebook。

```
cd \pywork
activate tensorflow-gpu jupyter notebook
```

```
命令提示符 - jupyter notebook
Microsoft Windows [版本 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\kevin>cd \pywork

C:\pywork>activate tensorflow-gpu

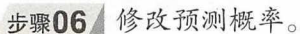
(tensorflow-gpu) C:\pywork>jupyter notebook
[I 15:08:31.424 NotebookApp] Serving notebooks from local
[I 15:08:31.440 NotebookApp] 0 active kernels
[I 15:08:31.440 NotebookApp] The Jupyter Notebook is running
```

步骤04 打开 TensorFlow Mnist CNN.ipynb，如图 21-19 所示。



当我们计算准确率时，测试数据共有 10 000 项，当传入全部的测试数据 `mnist.test.images`、`mnist.test.labels` 时，因为显卡内存只有 2GB，所以会发生错误，必须改为分两次执行，每次只传入 5000 项 `test` 数据。

- 传入前 5000 项数据：分别是 `mnist.test.images[:5000]`、`mnist.test.labels[:5000]`。
- 传入后 5000 项数据：分别是 `mnist.test.images[5000:]`、`mnist.test.labels[5000:]`。



因为显卡内存只有 2GB，所以我们在预测数据时，只能预测前 5000 项数据。

```
In [23]: y_predict=sess.run(y_predict,
                             feed_dict={x: mnist.test.images[:5000],
                                           y_label: mnist.test.labels[:5000],
                                           keep_probability: 1.})
```

步骤07 执行训练后的结果如图 21-20 所示。

```
i= 0, Loss = 803.555725, Training Accuracy = 0.11719
i= 100, Loss = 63.983269, Training Accuracy = 0.85156
i= 200, Loss = 43.454472, Training Accuracy = 0.91406
i= 300, Loss = 16.107862, Training Accuracy = 0.96875
i= 400, Loss = 19.796738, Training Accuracy = 0.94531
i= 500, Loss = 11.344000, Training Accuracy = 0.96875
i= 600, Loss = 10.940333, Training Accuracy = 0.98438
i= 700, Loss = 17.528381, Training Accuracy = 0.95312
i= 800, Loss = 8.724415, Training Accuracy = 0.97656
i= 900, Loss = 9.420431, Training Accuracy = 0.97656
i=1000, Loss = 7.894911, Training Accuracy = 0.98438
i=1100, Loss = 6.022167, Training Accuracy = 0.98438
i=1200, Loss = 9.873905, Training Accuracy = 0.96094
i=1300, Loss = 11.773293, Training Accuracy = 0.96875
i=1400, Loss = 8.008314, Training Accuracy = 0.96875
299.9933114051819
```

图 21-20

以上使用 GPU 执行训练所花费的时间是 299 秒，之前使用 CPU 执行训练所花费的时间是 395 秒，GPU 比 CPU 运行时间减少了 96 秒。大约计算了一下， $(96 \div 395) \times 100 = 24$ ，也就是说 GPU 比 CPU 大约减少 24% 的时间。

21.6 以 Keras Cifar CNN 的实际范例比较 CPU 与 GPU 的执行速度

之前测试的都是 TensorFlow 程序，接下来我们将测试 Keras 程序比较 CPU 与 GPU 的执行速度，将之前第 10 章所创建的 Keras_Cifar_CNN.ipynb 作为测试范例。

步骤01 启动 TensorFlow CPU 环境。

➤ 启动 jupyter notebook

```
cd \pywork activate tensorflow jupyter notebook
```

执行后屏幕显示界面如图 21-21 所示。

```

命令提示符 - jupyter notebook
Microsoft Windows [版本 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\kevin>cd \pywork

C:\pywork>activate tensorflow

(tensorflow) C:\pywork>jupyter notebook
[I 15:05:03.549 NotebookApp] Serving notebooks from local disk
[I 15:05:03.549 NotebookApp] 0 active kernels

```

图 21-21

步骤02 在 TensorFlow CPU 环境中执行训练。

进入 jupyter notebook 界面后，打开并执行 Keras_Cifar_CNN.ipynb 程序，执行训练的界面如图 21-22 所示。

```

In [24]: train_history=model.fit(x_img_train_normalize, y_label_train_OneHot,
                                validation_split=0.2,
                                epochs=10, batch_size=128, verbose=1)

Train on 40000 samples, validate on 10000 samples
Epoch 1/10
40000/40000 [=====] - 170s - loss: 1.5028 - acc: 0.4581
Epoch 2/10
40000/40000 [=====] - 169s - loss: 1.1428 - acc: 0.5952
Epoch 3/10
40000/40000 [=====] - 173s - loss: 0.9831 - acc: 0.6551
Epoch 4/10
40000/40000 [=====] - 173s - loss: 0.8785 - acc: 0.6912
Epoch 5/10
40000/40000 [=====] - 173s - loss: 0.7868 - acc: 0.7230

```

每一个训练周期的运行时间大约为 170 秒

图 21-22

以上每一个训练周期的运行时间大约为 170 秒。

步骤03 启动 TensorFlow GPU 环境。

关闭之前的“命令提示符”窗口，再打开新的“命令提示符”窗口，输入下列命令启动 TensorFlow GPU 环境，再启动 jupyter notebook。

```

cd \pywork
activate tensorflow-gpu jupyter notebook

```

执行后屏幕显示界面如图 21-23 所示。

```

命令提示符 - jupyter notebook
Microsoft Windows [版本 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\kevin>cd \pywork

C:\pywork>activate tensorflow-gpu

(tensorflow-gpu) C:\pywork>jupyter notebook
[I 15:08:31.424 NotebookApp] Serving notebooks from local
[I 15:08:31.440 NotebookApp] 0 active kernels
[I 15:08:31.440 NotebookApp] The Jupyter Notebook is running

```

图 21-23

步骤04 在 Anaconda TensorFlow CPU 环境中执行训练。

进入 jupyter notebook 界面后, 打开并执行 Keras_Cifar_CNN.ipynb 程序, 执行训练的界面如图 21-24 所示。

每一个训练周期的运行时间大约为 109 秒

```

In [24]: train_history=model.fit(x_img_train_normalize, y_label_train_OneHot,
                                validation_split=0.2,
                                epochs=10, batch_size=128, verbose=1)

```

```

Train on 40000 samples, validate on 10000 samples
Epoch 1/10
40000/40000 [=====] - 109s - loss: 1.5207 - acc: 0.4525
Epoch 2/10
40000/40000 [=====] - 104s - loss: 1.1305 - acc: 0.5956
Epoch 3/10
40000/40000 [=====] - 104s - loss: 0.9847 - acc: 0.6538
Epoch 4/10
40000/40000 [=====] - 104s - loss: 0.8776 - acc: 0.6915
Epoch 5/10
40000/40000 [=====] - 104s - loss: 0.7887 - acc: 0.7215

```

图 21-24

以上使用 GPU 执行每一个训练周期运行的时间大约是 109 秒, 之前使用 CPU 执行每一个训练周期大约是 170 秒, GPU 运行时间减少了大约 61 秒。大约计算了一下, $(61 \div 170) \times 100 = 35$, 也就是说 GPU 比 CPU 大约减少了 35% 的时间。

21.7 结论

经过测试 CPU 与 GPU 的运行时间, 会发现 GPU 的运行时间比 CPU 大约减少了 35%。你也许觉得节省这么一点时间实在微不足道, 但是笔者测试的 GPU 独立显卡是最入门级的 GeForce GT720, 当时的价格不到 450 元, 而使用的 CPU 是 Intel Core i5 (英特尔酷睿) 中央处理器, 同样的时间市价大约是 1300 元。也就是说在深度学习训练中, 450 元不到的 GPU 打败了 1300 元的 CPU, 所以还是很值得这么做的。如果安装了更好的 Nvidia 显卡, 显卡内存再多一些, 相信执行性能会好很多。

附录A

本书范例程序的下载与安装说明

本书范例程序的下载与安装说明将分别介绍在 Windows 与 Linux 系统中的安装方式。读者可以按照自己安装的操作系统来选择安装本书的范例程序。

A.1 在 Windows 系统中下载与安装范例程序

在 Windows 中下载与安装范例程序之前，先参照本书第 4 章的说明在 Windows 中安装 TensorFlow 与 Keras。

步骤01 下载范例程序。

在浏览器中输入下列网址来下载本书的范例程序（见图 A-1）：

<https://pan.baidu.com/s/1c2rXnH2>



图 A-1

步骤02 解压缩范例程序。

下载文件后，打开文件资源管理器，单击下载目录就可以看到 MP21710_example 范例程序压缩文件，如图 A-2 所示。在 Windows 文件资源管理器中，双击文件名即可打开压缩文件。如果安装了其他解压缩软件，也可以用其他解压缩软件来解压缩。



图 A-2

步骤03 拖曳到 D:磁盘驱动器。

打开压缩文件后, 就可以看到 MP21710_example 的目录, 将其拖曳到 D:驱动器, 就可以自动解压缩到 D:驱动器 (安装在 D:驱动器只是示范, 读者也可以安装在其他地方), 如图 A-3 所示。

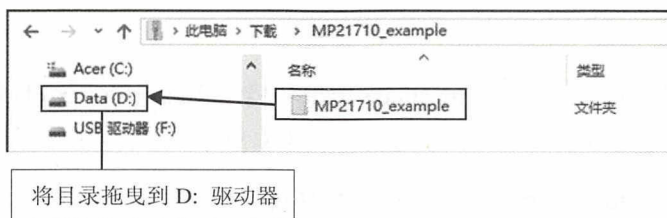


图 A-3

步骤04 查看 D:驱动器。

查看 D:驱动器, 我们可以看到已解压缩的范例程序目录, 如图 A-4 所示。

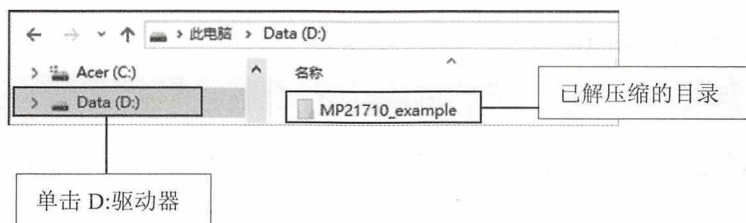


图 A-4

步骤05 启动 TensorFlow 的 Anaconda 虚拟环境。

打开“命令提示符”窗口, 输入下列命令。

➤ 切换到 D:驱动器

```
d:
```

➤ 切换到范例程序目录

```
cd \MP21710_example
```

➤ 启动 TensorFlow 的 Anaconda 虚拟环境

```
activate tensorflow
```

执行后屏幕显示界面如图 A-5 所示。



图 A-5

注意，以上启动 TensorFlow 的 Anaconda 虚拟环境是 TensorFlow CPU 版本。
如果读者参照本书第 20 章的介绍安装 TensorFlow GPU 版本，就使用下列指令。

➤ 启动 Anaconda TensorFlow GPU 虚拟环境

```
activate tensorflow-gpu
```

步骤06 打开 jupyter notebook。

打开“命令提示符”窗口，输入下列命令进入 jupyter notebook 互动界面。

➤ 打开 jupyter notebook

```
jupyter notebook
```

按 Enter 键后就会打开浏览器，默认的网址是 <http://localhost:8888>，即 jupyter notebook 界面，如图 A-6 所示。

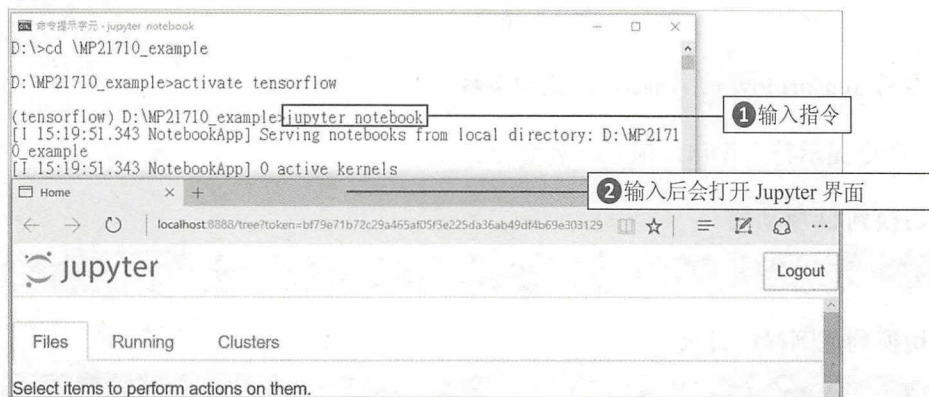


图 A-6

步骤07 打开 Jupyter 查看本书的范例程序。

打开 Jupyter 后，可以看到本书范例程序分为两大部分，分别是 Keras 与 TensorFlow 范例程序，如图 A-7 所示。



图 A-7

关于范例程序的功能说明可参考“本书章节与范例程序介绍”。

步骤08 查看本书范例程序目录。

打开 Jupyter 后，我们可以看到目录如图 A-8 所示。

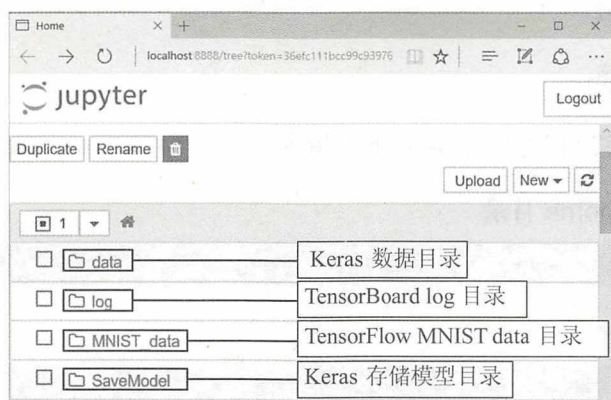


图 A-8

以上目录都是空的，必须执行范例程序后才会产生数据，说明如下。

➤ data: Keras 数据目录

用于存放 Keras 范例程序下载的数据文件，必须执行范例程序后，才会下载数据到此目录。

- 执行 Keras_Taianic_Introduce.ipynb 下载 Taianic 数据集后，才能执行其他 Taianic 程序。
- 执行 Keras_Imdb_Introduce.ipynb 下载并且解压缩 IMDB 数据集后，才能执行其他 IMDB 程序。

➤ log: TensorBoard log 目录

用于存放 TensorBoard 的 log 目录，必须执行范例程序 TensorFlow_Board_area.ipynb、TensorFlow_Mnist_CNN.ipynb 后，才会产生 log，并且在 TensorBoard 显示“计算图”。

➤ MNIST_data: TensorFlow MNIST data 目录

用于存放 TensorFlow MNIST 的数据文件，必须执行 TensorFlow_Mnist_Introduce.ipynb 才会下载文件。

➤ SaveModel: Keras 存储模型目录

用于存放 Keras 存储模型的目录，必须执行 Keras_Cifar_CNN_Continue_Train.ipynb 后才会产生。

以上范例的详细说明可参考本书对应的章节。

A.2 在 Ubuntu Linux 系统中下载与安装范例程序

在 Ubuntu Linux 下载与安装范例程序之前，先参照本书第 5 章的说明在 Linux Ubuntu 安装 TensorFlow 与 Keras。

步骤01 下载范例程序。

启动 Ubuntu 的“终端”程序，输入下列指令。

➤ 切换到用户 home 目录

```
cd ~/
```

➤ 下载范例程序

```
wget http://www.drmaster.com.tw/download/example/MP21710_example.zip
```

执行后屏幕显示界面如图 A-9 所示。

```
user@Ubuntu1604: ~  
user@Ubuntu1604:~$ cd ~/  
user@Ubuntu1604:~$ wget http://www.drmaster.com.tw/download/example/MP21710_example.zip  
--2017-05-12 11:38:15-- http://www.drmaster.com.tw/download/example/MP21710_example.zip  
正在解析主机 www.drmaster.com.tw (www.drmaster.com.tw)... 59.120.44.22  
正在连接 www.drmaster.com.tw (www.drmaster.com.tw)[59.120.44.22]:80... 已连接。  
已发出 HTTP 请求，正在等待回应... 200 OK  
长度: 2817520 (2.7M) [application/x-zip-compressed]  
Saving to: 'MP21710_example.zip'  
  
MP21710_example.zip 100%[=====] 2.69M 414KB/s in 6.6s  
2017-05-12 11:38:21 (414 KB/s) - 'MP21710_example.zip' saved [2817520/2817520]
```

图 A-9

步骤02 解压缩范例程序。

在“终端”程序中输入表 A-1 中的命令解压缩范例程序。

表 A-1 解压缩程序命令说明

命令	说明
ll MP21710_example.zip	查看下载的程序
unzip MP21710_example.zip	解压缩范例程序



执行后屏幕显示界面如图 A-10 所示。

```
user@Ubuntu1604: ~
user@Ubuntu1604:~$ ll MP21710_example.zip
-rw-rw-r-- 1 user user 2817520 5月 12 11:28 MP21710_example.zip
user@Ubuntu1604:~$ unzip MP21710_example.zip
Archive:  MP21710_example.zip
  creating: MP21710_example/
  creating: MP21710_example/.ipynb_checkpoints/
  inflating: MP21710_example/.ipynb_checkpoints/Keras_Imdb_Introduce-checkpoint.
  ipynb
  inflating: MP21710_example/.ipynb_checkpoints/Tensorflow_Mnist_Introduce-check
  point.ipynb
  creating: MP21710_example/data/
```

图 A-10

步骤03 打开 jupyter notebook。

在“终端”程序中输入下列命令进入 jupyter notebook 互动界面。

➤ 切换到范例程序目录

```
cd ~/MP21710_example
```

➤ 打开 jupyter notebook

```
jupyter notebook
```

按 Enter 键后就会打开浏览器，默认的网址是 <http://localhost:8888>，这是 Jupyter 界面，如图 A-11 所示。

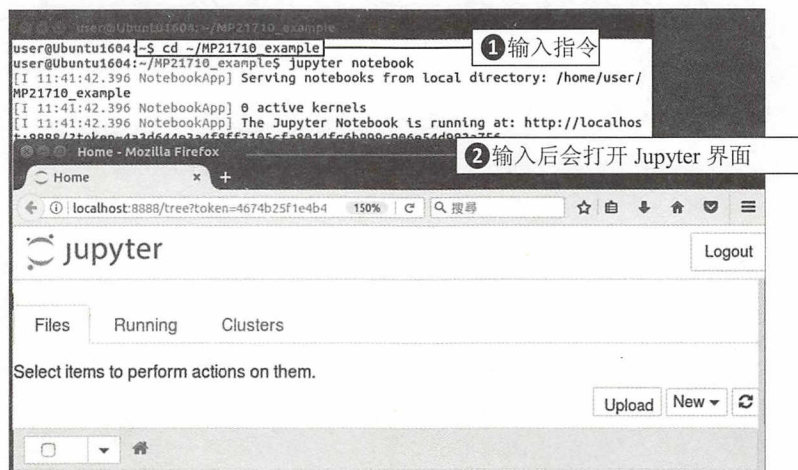


图 A-11

打开 Jupyter 后的目录与范例程序与在 Windows 系统下安装范例程序后的结果完全相同，可参考之前的说明。

TensorFlow + Keras

深度学习人工智能实践应用



轻松学会“深度学习”

先学习Keras再学习TensorFlow

TensorFlow 功能强大，执行效率高，支持各种平台，由于其是比较底层的深度学习链接库，学习门槛高，因此本书先介绍 Keras，这是以 TensorFlow 为底层的、高级的深度学习链接库，可以很容易地建立深度学习模型，进行训练并使用模型预测，对初学者而言学习门槛较低，在读者熟悉了深度学习模型后，再学习 TensorFlow 就很轻松了。

在 Windows 操作系统中安装 TensorFlow 1.0 与 Keras 2.0

对于初学者而言，在 Windows 操作系统中安装 TensorFlow 与 Keras 非常简单、容易上手。本书以详细的步骤说明如何在 Windows 操作系统中安装 TensorFlow 1.0 与 Keras 2.0。

在 Linux Ubuntu 操作系统中安装 TensorFlow 1.0 与 Keras 2.0

因为 Linux 操作系统是大数据分析 with 机器学习很常用的平台，所以本书以详细的步骤说明如何在 Linux Ubuntu 操作系统中安装 TensorFlow 1.0 与 Keras 2.0。

使用 GPU 大幅加快深度学习训练

GPU 提供了强大的并行计算架构，可让深度学习的训练比普通 CPU 快数十倍，读者只需要有 Nvidia 显卡，然后按照本书的步骤说明安装 CUDA、cuDNN、TensorFlow GPU 版本与 Keras，就可以使用 GPU 大幅加快深度学习的训练。

MNIST 手写数字识别，可识别 0~9 的手写数字

以实际范例说明如何使用 Keras 与 TensorFlow 构建多层感知器（MLP）、卷积神经网络（CNN）模型，可识别 0~9 的手写数字。

CIFAR-10 照片图像物体识别，可识别 10 种物体

以实际范例说明如何使用 Keras 构建多层感知器模型，可以识别飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船、卡车 10 种物体。

预测泰坦尼克号上旅客的生存概率

以实际范例说明如何使用 Keras 构建多层感知器模型，预测泰坦尼克号上的旅客及《泰坦尼克号》电影男女主角的生存概率，并找出泰坦尼克号上其他旅客的感人故事。

IMDb 影评文字自然语言处理与情感分析

情感分析的商业价值在于通过文字分析可提早得知顾客对公司或产品的观感，以调整销售策略的方向。本书以实际范例说明如何使用 Keras 构建多层感知器模型、递归神经网络（RNN）、长短时记忆（LSTM）等模型，以及预测影评文字是正面还是负面的评价。

清华社官方微信号



扫 我 有 惊 喜

ISBN 978-7-302-49302-0



9 787302 493020 >

定价：69.00 元